Quotient Types

Peter V. Homeier

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

Abstract. The quotient operation is a standard feature of set theory, where a set is divided into subsets by an equivalence relation; the resulting subsets of equivalent elements are called equivalence classes. 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 also to define functions that map between the original type and the quotient type. Several properties of these mapping functions are proven automatically. Tools are also provided to create quotients of aggregate types, in particular lists, pairs, and sums. These require special attention to preserve the aggregate structure across the quotient operation. We demonstrate these tools on an example from the sigma calculus.

1 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 [3].

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 [3] that

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

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

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

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

A *partition* Π of a set S is a set of nonempty subsets of S that is disjoint and exhaustive. Then $S/\sim$ is a partition of S [3].

2 Quotient Types

Now let us reinterpret this for the Higher Order Logic theorem prover, whose logic is a type theory, rather than a set theory. The following definitions and proofs are accomplished within HOL [4]. Let τ be any type; then a binary relation on τ can be represented in HOL as a curried function of type $\tau \to (\tau \to \texttt{bool})$. Let E be an equivalence relation, which is a binary relation satisfying

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

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

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

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

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

Proof: We prove this as a biconditional.

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

Case 2. ($\Leftarrow$) Assume $E\ x\ y$. We must show that $[x]_E = [y]_E$, which by definition 1 is $E\ x = E\ y$. This is equality between functions, which by extension follows from $\forall z.\ E\ x\ z \Leftrightarrow E\ y\ z$. We prove this as a biconditional.

Case 2.1. ($\Rightarrow$) Assume $E\ x\ z$. Since $E\ x\ y$, $E\ y\ x$ follows by symmetry and then $E\ y\ z$ follows by transitivity.

Case 2.2. ($\Leftarrow$) Assume $E\ y\ z$. Since $E\ x\ y$, $E\ x\ z$ follows by transitivity. □

New types may be defined in HOL using the function `new_type_definition` [4, sections 18.2.2.3-5]. This function requires us to choose a representing type, and a predicate on that type denoting a subset that is nonempty.

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

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

P is nonempty because $P\ [x]_E$ for all $x : \tau$. Let $\tau' = \tau/E$. We then use the HOL tool `define_new_type_bijections` [4], which automatically defines a bijection and its inverse, $\lfloor \_ \rfloor_c : (\tau \to \texttt{bool}) \to \tau'$ and $\lceil \_ \rceil_c : \tau' \to (\tau \to \texttt{bool})$, such that

Theorem 4. $(\forall a : \tau'.\ \lfloor \lceil a \rceil_c \rfloor_c = a)\ \wedge\ (\forall f : \tau \to \texttt{bool}.\ P\ f \Leftrightarrow (\lceil \lfloor f \rfloor_c \rceil_c = f))$

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

Theorem 5. $(\forall a : \tau'.\ \lfloor \lceil a \rceil_c \rfloor_c = a)\ \wedge\ (\forall r : \tau.\ \lceil \lfloor [r]_E \rfloor_c \rceil_c = [r]_E)$.

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

Lemma 6 ($\lceil \_ \rceil_c$ is onto equivalence classes). $\forall a.\ \exists r.\ \lceil a \rceil_c = [r]_E$

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

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

Proof: We will prove this as a biconditional.

Case 1.($\Rightarrow$) Assume $\lfloor [x]_E \rfloor_c = \lfloor [y]_E \rfloor_c$. Taking the $\lceil \_ \rceil_c$ of both sides gives us $\lceil \lfloor [x]_E \rfloor_c \rceil_c = \lceil \lfloor [y]_E \rfloor_c \rceil_c$. By the right clause of theorem 5, $[x]_E = [y]_E$.

Case 2.($\Leftarrow$) Assume $[x]_E = [y]_E$. Then applying $\lfloor \_ \rfloor_c$ to both sides yields the goal, $\lfloor [x]_E \rfloor_c = \lfloor [y]_E \rfloor_c$. □

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

Definition 8 (Quotient maps).

$$\begin{array}{lllll} \lfloor \_ \rfloor : \tau \to \tau' & \lfloor x \rfloor & \stackrel{\text{def}}{=} & \lfloor [x]_E \rfloor_c & (\ = \ \lfloor E\ x \rfloor_c\ ) \\ \lceil \_ \rceil : \tau' \to \tau & \lceil a \rceil & \stackrel{\text{def}}{=} & \texttt{\$@}\ \lceil a \rceil_c & (\ = \ \texttt{@}x.\ \lceil a \rceil_c\ x \ = \ \texttt{@}x.\ ([x]_E = \lceil a \rceil_c)\ ) \end{array}$$

The @ operator, which is here used as a prefix unary operator $, is a higher order version of Hilbert's choice operator ε [4]. It has type $(\alpha \to \texttt{bool}) \to \alpha$, and is usually used as a binder, where $\texttt{\$@}\ t = \texttt{@}x.\ t\ x$. Given a predicate Q on a type, if any element of the type satisfies the predicate, then $\texttt{\$@}\ Q$ returns an arbitrary element of that type which satisfies Q. If no element of the type satisfies Q, then $\texttt{\$@}\ Q$ will return simply some unknown, arbitrary element of the type. The axiom in HOL about the behavior of @ is $\forall t\ x.\ t\ x \Rightarrow t\ (\texttt{\$@}\ t)$. In the definition above, it selects some arbitrary representative x such that the equivalence class of x is the class representating a.

Lemma 9. $\forall r.\ [\texttt{\$@}\ [r]_E\ ]_E = [r]_E$

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

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

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

$$\begin{array}{lcll} \lfloor \lceil a \rceil \rfloor & = & \lfloor [\texttt{\$@}\ \lceil a \rceil_c\ ]_E \rfloor_c & \text{definition 8} \\ & = & \lfloor [\texttt{\$@}\ [r]_E\ ]_E \rfloor_c & \text{selection of } r \\ & = & \lfloor [r]_E \rfloor_c & \text{lemma 9} \\ & = & \lfloor \lceil a \rceil_c \rfloor_c & \text{selection of } r \\ & = & a & \text{theorem 4} \end{array}$$

□

Theorem 11. $\forall r\ r'.\ \ E\ r\ r' \ \Leftrightarrow\ (\lfloor r \rfloor = \lfloor r' \rfloor)$

Proof:

$$\begin{array}{lll} E\ r\ r' & \Leftrightarrow & [r]_E = [r']_E \qquad \text{lemma 2} \\ & \Leftrightarrow & \lfloor [r]_E \rfloor_c = \lfloor [r']_E \rfloor_c \qquad \text{lemma 7} \\ & \Leftrightarrow & \lfloor r \rfloor = \lfloor r' \rfloor \qquad \text{definition 8} \end{array}$$

□

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

3 New Quotient Type Definitions

In this section we describe how to use the main tool for defining quotient types, the function `define_quotient_type`. This function automates the reasoning described in the last section, creating the quotient type as a new type in the HOL logic, and also defining the mapping functions between the types, relating them as described in theorems 10 and 11.

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

Before invoking the quotient package, the user should define an equivalence relation on the original type `ty`. The equivalence relation should be expressed as a constant *EQUIV*, of type `ty -> ty -> bool`. The user should prove that *EQUIV* is indeed an equivalence relation, by proving three theorems of the following forms:

reflexivity	`|- !a.` *EQUIV* `a a`
symmetry	`|- !a b.` *EQUIV* `a b =` *EQUIV* `b a`
transitivity	`|- !a b c.` *EQUIV* `a b /\` *EQUIV* `b c ==>` *EQUIV* `a c`

Then the new quotient type may be constructed by the following function.

```
define_quotient_type : string -> string -> string ->
      thm -> thm -> thm -> thm
```

Evaluating

```
define_quotient_type "tyname" "abs" "rep"
      |- !a:ty. EQUIV a a
      |- !a b:ty. EQUIV a b = EQUIV b a
      |- !a b c:ty. EQUIV a b /\ EQUIV b c ==> EQUIV a c
```

automatically declares a new type *tyname* in the HOL logic as the quotient type `ty`/*EQUIV*, which we will refer to from here on as `newty`, and declares two new constants *abs*`:ty->newty` and *rep*`:newty->ty`, such that

```
|- (!a. abs(rep a) = a) /\ (!r r'. EQUIV r r' = (abs r = abs r'))
```

This theorem, which could be the defining property for the constants *abs* and *rep*, is stored in the current theory under the automatically-generated name *tyname*_QUOTIENT. It is also the value returned by `define_quotient_type`. This theorem states that *abs* is the left inverse of *rep*, and that *abs* maps equivalence between values of `ty` to equality between values of `newty`. Theorems of this form are much used in this package. Since they are often referred to, we give them the special name of "quotient theorems."

Observe that *abs* and *rep* are total functions, and are one-to-one and onto, up to equivalence. For any element of `ty`, *abs* maps it to the element in `newty` which corresponds to the equivalence class containing that element. Also, for any element of `newty`, *rep* maps it to an element of `ty` which is some element of the corresponding equivalence class. We do not know exactly which one is chosen, but we do know that since *rep* is a function, it yields the same representative element each time for the same element of `newty`.

The three theorem arguments concerning reflexivity, symmetry, and transitivity are equivalent to a single theorem of the form

```
|- !a b:ty. (EQUIV a = EQUIV b) = EQUIV a b
```

4 Properties of Quotient Types

Given a quotient theorem, there are several tools provided in the package that prove and return useful derivative results.

```
prove_quotient_rep_abs_equiv        : thm -> thm
prove_quotient_rep_abs_equal        : thm -> thm
prove_quotient_rep_abs_equal2       : thm -> thm
prove_quotient_equal_is_rep_equiv   : thm -> thm
prove_quotient_equiv_rep_one_one    : thm -> thm
prove_quotient_abs_iso_equiv        : thm -> thm
prove_quotient_rep_fn_onto          : thm -> thm
prove_quotient_abs_fn_onto          : thm -> thm
prove_quotient_equiv_equal          : thm -> thm
prove_quotient_equiv_refl           : thm -> thm
prove_quotient_equiv_sym            : thm -> thm
prove_quotient_equiv_trans          : thm -> thm
```

Each of these takes as its argument a quotient theorem of the form returned by `define_quotient_type`:

```
|- (!a. abs(rep a) = a) /\ (!r r'. EQUIV r r' = (abs r = abs r'))
```

If *th* is of this form, then evaluating `prove_quotient_rep_abs_equiv` *th* proves that *rep* is the left inverse of *abs*, up to equivalence, returning the theorem:

```
|- !r. EQUIV (rep (abs r)) r
```

Evaluating `prove_quotient_rep_abs_equal` *th* proves that the composition of *rep* with *abs* of a value has the same equivalence class as the original value:

```
|- !r. EQUIV (rep (abs r)) = EQUIV r
```

Evaluating `prove_quotient_rep_abs_equal2` *th* proves a symmetric version of the same idea, both of which are useful for rewriting:

```
|- !r r'. EQUIV r (rep (abs r')) = EQUIV r r'
```

Evaluating `prove_quotient_equal_is_rep_equiv` *th* proves that equality between abstract values is equivalence between representatives:

```
|- !a a'. (a = a') = EQUIV (rep a) (rep a')
```

Evaluating `prove_quotient_rep_fn_one_one` *th* proves that *rep* is one-to-one:

```
|- !a a'. (rep a = rep a') = (a = a')
```

Evaluating `prove_quotient_abs_iso_equiv` *th* proves that *abs* and *EQUIV* are isomorphic with respect to equality:

```
|- !r r'. (abs r = abs r') = (EQUIV r = EQUIV r')
```

Evaluating `prove_quotient_rep_fn_onto` *th* proves that *rep* is onto, up to equivalence:

```
|- !r. ?a. EQUIV r (rep a)
```

Evaluating `prove_quotient_abs_fn_onto` *th* proves that *abs* is onto:

```
|- !a. ?r. a = abs r
```

Evaluating `prove_quotient_equiv_equal` *th* proves that equality between equivalence classes is equivalence between elements:

```
|- !x y. (EQUIV x = EQUIV y) = EQUIV x y
```

Evaluating `prove_quotient_equiv_refl` *th* proves that *EQUIV* is reflexive:

```
|- !x. EQUIV x x
```

Evaluating `prove_quotient_equiv_sym` *th* proves that *EQUIV* is symmetric:

```
|- !x y. EQUIV x y = EQUIV y x
```

And evaluating `prove_quotient_equiv_trans` *th* proves that *EQUIV* is transitive:

```
|- !x y z. EQUIV x y /\ EQUIV y z ==> EQUIV x z
```

The last three functions are useful when the quotient theorem *th* is derived from another source than `define_quotient_type`.

None of these functions saves anything on the current theory file. In fact, it should usually be unnecessary to save the results proved by these functions, since they can be generated quickly whenever required from the theorem returned by `define_quotient_type`, which is is itself saved.

5 Aggregate Quotient Types

At times one wishes to take the quotients of a family of related types in parallel. This can be done by applying `define_quotient_type` repeatedly to each of the types. This is fine if the original types are independent of each other. However, if these types are related structurally, where for example some of them are formed as lists or pairs of other types in the family, then that structural information would be forgotten by this approach.

If we were intending to recreate the logical structure that existed among the original types, modulo the equivalence relations, then we have not precisely succeeded. If we use `define_quotient_type` to naïvely create the quotients of the list and pair types, then the resulting new types are not themselves in fact list or pair types. It turns out that they are *isomorphic* to list and pair types, but they are not *identical*. Since they are not recognized within HOL as list and pair types, we lose all connection to the exiting body of theory within HOL that is available to treat lists and pairs, and we must laboriously reconstruct what portions we need for these new quotient types.

To avoid this, the quotient package offers several functions for constructing the quotients of list types, pair types, and sum types.

```
new_list_equiv_and_quotient_maps
new_pair_equiv_and_quotient_maps
new_sum_equiv_and_quotient_maps

new_equiv_and_quotient_maps

new_list_quotient_maps
new_pair_quotient_maps
new_sum_quotient_maps

new_quotient_maps
```

These functions define the mapping functions between the old and new types, and prove key theorems about their behavior. These functions do not create new types, since the types are formed using the `list`, `prod`, and `sum` type operators.

The first four functions listed also define as a new constant the equivalence relation for the aggregate type (list, pair, or sum). The last four expect the equivalence relation to be previously defined, and take as an argument a theorem about its value on all the ways the aggregate type can be constructed.

`new_equiv_and_quotient_maps` is a uniform interface to the first three functions, and `new_quotient_maps` is a uniform interface for the three before it.

These functions all take as arguments quotient theorems relating to the types of the component elements of the aggregate type. Sometimes one desires to perform the quotient operation on some components but not on others. In these cases, to indicate a component which is not to be divided, the argument provided should be instead the standard theorem `TRUTH`, `|- T`.

Each of the first four functions returns a triple of three theorems. The first theorem of the triple is a definition of the equivalence relation which has been defined, for each combination of the constructors of the aggregate type. The second theorem of the triple describes the definition of the abstraction and representation functions, mapping between the new and old types, in terms of the abstraction and representation functions of the respective element types. The third theorem of the triple is a quotient theorem, of the same form as that returned by `define_quotient_type`, relating to the aggregate type.

Each of the last four functions listed returns a pair of two theorems, which are the same as the second and third theorems of the triple described above.

All of the theorems returned by these functions are saved in the current theory under automatically-generated names.

It is the presence of the theorem that defines the mapping functions that provides additional information about the mapping functions, beyond that available using `define_quotient_type`. This structure, present among the original types, might have been lost among the new types by the naïve approach. The structure would have been there implicitly, but not explicitly recognized within HOL as lists, pairs, and sums.

5.1 List quotients by a new equivalence relation

To define an equivalence relation between lists based on an equivalence relation on the elements, and then use it to define the mapping functions between types of lists of original and quotient types, the following ML function is provided:

```
new_list_equiv_and_quotient_maps :
      string -> string -> string -> string ->
      hol_type -> thm -> (thm # thm # thm)
```

Evaluating

`new_list_equiv_and_quotient_maps` "*name*" "*EQUIV*" "*abs*" "*rep*" *list-type elem-quotient-thm*

where *list-type* is of the form `(ty)list`, and where *elem-quotient-thm* is a quotient theorem

```
|- (!a:newty. elem-abs(elem-rep a) = a) /\
   (!r r':ty. EQUIV-elem r r' = (elem-abs r = elem-abs r'))
```

automatically defines an equivalence relation
EQUIV: `(newty)list -> (newty)list -> bool` such that

```
|- (!e1 e2 l1 l2.
     EQUIV (e1::l1) (e2::l2) =
     EQUIV-elem e1 e2 /\ EQUIV l1 l2) /\
   (EQUIV [] [] = T) /\
   (!e l. EQUIV (e::l) [] = F) /\
   (!e l. EQUIV [] (e::l) = F)
```

and two new constants *abs*:(ty)list->(newty)list and *rep*:(newty)list->(ty)list such that:

```
|- ((abs [] = []) /\
    (!e l. abs (e::l) = elem-abs e :: abs l)) /\
   ((rep [] = []) /\
    (!e l. rep (e::l) = elem-rep e :: rep l))
|- (!a. abs(rep a) = a) /\
   (!r r'. EQUIV r r' = (abs r = abs r'))
```

and returns a triple of these three theorems. Note that the third is a quotient theorem for the list type. These theorems are stored in the current theory under the names *EQUIV*_DEF, *name*_MAPS_DEF, and *name*_QUOTIENT, respectively.

5.2 List quotients by an existing equivalence relation

To define the mapping functions between types of lists of original and quotient types, using an already-defined equivalence relation, the following ML function is provided:

```
new_list_quotient_maps :
      string -> string -> string -> thm -> thm -> (thm # thm)
```

Evaluating

```
new_list_quotient_maps "name" "abs" "rep"
       list-def-thm elem-quotient-thm
```

where *list-def-thm* is of the form:

```
|- (!e1 e2 l1 l2.
     EQUIV (e1::l1) (e2::l2) =
     EQUIV-elem e1 e2 /\ EQUIV l1 l2) /\
   (EQUIV [] [] = T) /\
   (!e l. EQUIV (e::l) [] = F) /\
   (!e l. EQUIV [] (e::l) = F)
```

and where *elem-quotient-thm* is a quotient theorem

```
|- (!a:newty. elem-abs(elem-rep a) = a) /\
   (!r r':ty. EQUIV-elem r r' = (elem-abs r = elem-abs r'))
```

automatically defines two new constants *abs*:(ty)list->(newty)list and *rep*:(newty)list->(ty)list such that:

```
|- ((abs [] = []) /\
    (!e l. abs (e::l) = elem-abs e :: abs l)) /\
   ((rep [] = []) /\
    (!e l. rep (e::l) = elem-rep e :: rep l))
|- (!a. abs(rep a) = a) /\
   (!r r'. EQUIV r r' = (abs r = abs r'))
```

and returns a pair of these two theorems. Note that the second is a quotient theorem for the list type. These theorems are stored in the current theory under the names *name*_MAPS_DEF and *name*_QUOTIENT, respectively.

5.3 Pair quotients by a new equivalence relation

To define an equivalence relation between pairs based on equivalence relations between the elements, and then use it to define the mapping functions between old and new types of pairs of quotient types, the following ML function is provided:

```
new_pair_equiv_and_quotient_maps :
      string -> string -> string -> string ->
      hol_type -> thm -> thm -> (thm # thm # thm)
```

Evaluating

```
new_pair_equiv_and_quotient_maps "name" "EQUIV" "abs" "rep"
      pair-type left-quotient-thm right-quotient-thm
```

where *pair-type* is of the form `(ty1 # ty2)`, and where *left-quotient-thm* is either `|- T` or a quotient theorem of the form

```
|- (!a:newty1. left-abs(left-rep a) = a) /\
   (!r r':ty1. EQUIV-left r r' = (left-abs r = left-abs r'))
```

and where *right-quotient-thm* is either `|- T` or a quotient theorem of the form

```
|- (!a:newty2. right-abs(right-rep a) = a) /\
   (!r r':ty2. EQUIV-right r r' = (right-abs r = right-abs r'))
```

automatically defines an equivalence relation
EQUIV: `(newty1 # newty2) -> (newty1 # newty2) -> bool` such that

```
|- !a1 a2 b1 b2.
     EQUIV (a1,b1) (a2,b2) =
     EQUIV-left a1 a2 /\ EQUIV-right b1 b2
```

and two new constants *abs*:`(ty1 # ty2)->(newty1 # newty2)` and
rep:`(newty1 # newty2)->(ty1 # ty2)` such that:

```
|- (!a b. abs (a,b) = (left-abs a,right-abs b)) /\
   (!a b. rep (a,b) = (left-rep a,right-rep b))
|- (!a. abs(rep a) = a) /\
   (!r r'. EQUIV r r' = (abs r = abs r'))
```

(or simpler versions of these theorems, depending on which elements of the pair are being divided) and returns a triple of these three theorems. Note that the third is a quotient theorem for the pair type. These theorems are stored in the current theory under names *EQUIV*_DEF, *name*_MAPS_DEF, and *name*_QUOTIENT, respectively.

5.4 Pair quotients by an existing equivalence relation

To define the mapping functions between types of pairs of original and quotient types, the following ML function is provided:

```
new_pair_quotient_maps :
      string -> string -> string ->
      thm -> thm -> thm -> (thm # thm)
```

Evaluating

```
new_pair_quotient_maps "name" "abs" "rep"
      pair-def-thm left-quotient-thm right-quotient-thm
```

where *pair-def-thm* is of the form

```
|- !a1 a2 b1 b2.
     EQUIV (a1,b1) (a2,b2) =
     EQUIV-left a1 a2 /\ EQUIV-right b1 b2
```

and where *left-quotient-thm* is either `|- T` or a quotient theorem of the form

```
|- (!a:newty1. left-abs(left-rep a) = a) /\
   (!r r':ty1. EQUIV-left r r' = (left-abs r = left-abs r'))
```

and where *right-quotient-thm* is either `|- T` or a quotient theorem of the form

```
|- (!a:newty2. right-abs(right-rep a) = a) /\
   (!r r':ty2. EQUIV-right r r' = (right-abs r = right-abs r'))
```

automatically defines two new constants *abs*`:(ty1 # ty2)->(newty1 # newty2)` and *rep*`:(newty1 # newty2)->(ty1 # ty2)` such that:

```
|- (!a b. abs (a,b) = (left-abs a,right-abs b)) /\
   (!a b. rep (a,b) = (left-rep a,right-rep b))
|- (!a. abs(rep a) = a) /\
   (!r r'. EQUIV r r' = (abs r = abs r'))
```

(or simpler versions of the theorems, depending on which elements of the pair are being divided) and returns a pair of these two theorems. Note that the second is a quotient theorem for the pair type. These theorems are stored in the current theory under the names *name*`_MAPS_DEF` and *name*`_QUOTIENT`, respectively.

5.5 Sum quotients

Quotients of sum types are handled in a way precisely analagous to that of pair types, as above, with the exception that the equivalence relation has the type *EQUIV*`: (newty1 + newty2) -> (newty1 + newty2) -> bool`, and the theorem defining the equivalence relation between sums has the form:

```
|- (!a a'. EQUIV (INL a) (INL a') = EQUIV-left a a') /\
   (!b b'. EQUIV (INR b) (INR b') = EQUIV-right b b') /\
   (!a b. EQUIV (INL a) (INR b) = F) /\
   (!a b. EQUIV (INR b) (INL a) = F)
```

and the theorem defining the mapping functions has the form:

```
|- ((!a. abs (INL a) = INL (left-abs a)) /\
    (!b. abs (INR b) = INR (right-abs b))) /\
   ((!a. rep (INL a) = INL (left-rep a)) /\
    (!b. rep (INR b) = INR (right-rep b))) .
```

5.6 Uniform interfaces for aggregate quotients

The following ML functions are provided as uniform interfaces to the previous aggregate quotient tools, where the quotient theorems for the components are gathered into a list argument:

```
new_equiv_and_quotient_maps :
   string -> string -> string -> string ->
   hol_type -> thm list -> (thm # thm # thm)
new_quotient_maps :
   string -> string -> string ->
   thm -> thm list -> (thm # thm)
```

Depending on the particular type, the appropriate specific function is called.

6 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.

7 The Pre-Sigma Calculus in HOL

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

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

```
val _ = Hol_datatype

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

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

This creates the new mutually recursive types `obj1` and `method1`, and more also.

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.

8 The Pure Sigma Calculus in HOL

We define the pure sigma calculus in steps, with the results of some steps being used in later steps. Specifically, the quotient theorem created for methods is used to create the quotient theorem for entries, which is itself then used to create the quotient theorem for dictionaries.

Let us assume that we have defined alpha-equivalence relations for each of the four types `obj1`, `dict1`, `entry1`, and `method1`, called `ALPHA_obj`, `ALPHA_dict`, `ALPHA_entry`, and `ALPHA_method`. Let us further assume that we have proven the reflexivity, symmetry, and transitivity of each of these, as theorems called `ALPHA_obj_REFL`, `ALPHA_obj_SYM`, `ALPHA_obj_TRANS`, et cetera. Finally, we assume that we have proven that the alpha-equivalence of `dict1` and `entry1` can be expressed in terms of the alpha-equivalence of their components, as theorems

```
ALPHA_dict_DEF
    |- (!e1 e2 d1 d2.
          ALPHA_dict (e1::d1) (e2::d2) =
          ALPHA_entry e1 e2 /\ ALPHA_dict d1 d2) /\
       (ALPHA_dict [] [] = T) /\
       (!e d. ALPHA_dict (e::d) [] = F) /\
       (!e d. ALPHA_dict [] (e::d) = F)

ALPHA_entry_DEF
    |- !l1 l2 m1 m2.
         ALPHA_entry (l1,m1) (l2,m2) =
         (l1 = l2) /\ ALPHA_method m1 m2
```

We first define the pure sigma calculus types `obj` and `method`:

```
- val obj_QUOTIENT =
    define_quotient_type "obj" "obj_ABS" "obj_REP"
            ALPHA_obj_REFL ALPHA_obj_SYM ALPHA_obj_TRANS;
> val obj_QUOTIENT =
    |- (!a. obj_ABS (obj_REP a) = a) /\
       (!r r'. ALPHA_obj r r' = (obj_ABS r = obj_ABS r')) : thm

- val method_QUOTIENT =
    define_quotient_type "method" "method_ABS" "method_REP"
            ALPHA_method_REFL ALPHA_method_SYM ALPHA_method_TRANS;
> val method_QUOTIENT =
    |- (!a. method_ABS (method_REP a) = a) /\
       (!r r'. ALPHA_method r r' = (method_ABS r = method_ABS r'))
    : thm
```

Now, since the equivalence relations for `entry1` and `dict1` are previously defined, we will use the tools that make use of their definitions.

We next define the abstraction and representation functions for the `entry = (string # method)` type as follows:

```
- val (entry_MAPS_DEF, entry_QUOTIENT) =
    new_pair_quotient_maps "entry" "entry_ABS" "entry_REP"
                  ALPHA_entry_DEF TRUTH method_QUOTIENT;
> val entry_MAPS_DEF =
    |- (!a b. entry_ABS (a,b) = (a,method_ABS b)) /\
       (!a b. entry_REP (a,b) = (a,method_REP b)) : thm

  val entry_QUOTIENT =
    |- (!a. entry_ABS (entry_REP a) = a) /\
       (!r r'. ALPHA_entry r r' = (entry_ABS r = entry_ABS r'))
    : thm
```

Now we can define the abstraction and representation functions for the `dict = (entry)list` type as follows:

```
- val (dict_MAPS_DEF, dict_QUOTIENT) =
    new_list_quotient_maps "dict" "dict_ABS" "dict_REP"
                ALPHA_dict_DEF entry_QUOTIENT;
> val dict_MAPS_DEF =
    |- ((dict_ABS [] = []) /\
        (!e l. dict_ABS (e::l) = entry_ABS e::dict_ABS l)) /\
       ((dict_REP [] = []) /\
        (!e l. dict_REP (e::l) = entry_REP e::dict_REP l)) : thm
  val dict_QUOTIENT =
    |- (!a. dict_ABS (dict_REP a) = a) /\
       (!r r'. ALPHA_dict r r' = (dict_ABS r = dict_ABS r')) : thm
```

If the `ALPHA_entry` and `ALPHA_dict` relations were previously undefined, then the `ALPHA_entry` relation could have been defined using `method_QUOTIENT`.

```
- val (ALPHA_entry_DEF, entry_MAPS_DEF, entry_QUOTIENT) =
    new_pair_equiv_and_quotient_maps
                "entry" "ALPHA_entry" "entry_ABS" "entry_REP"
                (==`:string # method1`==) TRUTH method_QUOTIENT;
> val ALPHA_entry_DEF =
    |- !a1 b1 a2 b2.
         ALPHA_entry (a1,b1) (a2,b2) =
         (a1 = a2) /\ ALPHA_method b1 b2 : thm
  val entry_MAPS_DEF =
    |- (!a b. entry_ABS (a,b) = (a,method_ABS b)) /\
       (!a b. entry_REP (a,b) = (a,method_REP b)) : thm
  val entry_QUOTIENT =
    |- (!a. entry_ABS (entry_REP a) = a) /\
       (!r r'. ALPHA_entry r r' = (entry_ABS r = entry_ABS r'))
    : thm
```

Then we would define the ALPHA_dict relation using entry_QUOTIENT.

```
- val (ALPHA_dict_DEF, dict_MAPS_DEF, dict_QUOTIENT) =
    new_list_equiv_and_quotient_maps
             "dict" "ALPHA_dict" "dict_ABS" "dict_REP"
             (==`:(string # method1)list`==) entry_QUOTIENT;
> val ALPHA_dict_DEF =
    |- (!e1 l1 e2 l2.
          ALPHA_dict (e1::l1) (e2::l2) =
          ALPHA_entry e1 e2 /\ ALPHA_dict l1 l2) /\
       (ALPHA_dict [] [] = T) /\
       (!e1 l1. ALPHA_dict (e1::l1) [] = F) /\
       (!e2 l2. ALPHA_dict [] (e2::l2) = F) : thm
  val dict_MAPS_DEF =
    |- ((dict_ABS [] = []) /\
        (!e l. dict_ABS (e::l) = entry_ABS e::dict_ABS l)) /\
       ((dict_REP [] = []) /\
        (!e l. dict_REP (e::l) = entry_REP e::dict_REP l)) : thm
  val dict_QUOTIENT =
    |- (!a. dict_ABS (dict_REP a) = a) /\
       (!r r'. ALPHA_dict r r' = (dict_ABS r = dict_ABS r')) : thm
```

Constructors for the quotient types can be defined using the mapping functions: $\texttt{OVAR}\ x \stackrel{\text{def}}{=} \lfloor \texttt{OVAR1}\ x \rfloor_o$, $\texttt{OBJ}\ b \stackrel{\text{def}}{=} \lfloor \texttt{OBJ1}\ \lceil b \rceil_d \rfloor_o$, $\texttt{SIGMA}\ x\ a \stackrel{\text{def}}{=} \lfloor \texttt{SIGMA1}\ x\ \lceil a \rceil_o \rfloor_m$, etc. Now we have $\texttt{SIGMA}\ x\ (\texttt{OVAR}\ x) = \texttt{SIGMA}\ y\ (\texttt{OVAR}\ y)$, etc., as intended. The pure sigma calculus is thus accomplished by identifying alpha-equivalent terms.

9 Conclusions

We have implemented a package for mechanically defining quotient types which is a conservative, definitional extension of the HOL logic.

Quotients are useful in a variety of contexts, as shown in the literature.

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

For an extended example of the use of this package in a significant proof, and for more information on these and other new HOL tools, please see [5].

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. Enderton, H. B.: *Elements of Set Theory.* Academic Press, 1977.
4. Gordon, M. J. C., Melham, T. F.: *Introduction to HOL.* Cambridge University Press, Cambridge, 1993.
5. Homeier, P. V.: `http://www.cis.upenn.edu/~hol/lamcr`.