Quotient Types

Peter Vincent Homeier
U.S. Department of Defense
homeier@saul.cis.upenn.edu
Quotient Sets

• A standard construction of set theory
• Based on equivalence relations: a relation \( \sim \) must be

  reflexive: \quad \forall x \in S. \quad x \sim x
  symmetric: \quad \forall x, y \in S. \quad x \sim y \Rightarrow y \sim x
  transitive: \quad \forall x, y, z \in S. \quad x \sim y \land y \sim z \Rightarrow x \sim z

• Equivalence class of \( x \): \quad [x]_\sim = \{ y \mid x \sim y \}
  – Then it follows that \quad [x]_\sim = [y]_\sim \iff x \sim y

• Quotient set \( S / \sim \): \quad S / \sim = \{ [x]_\sim \mid x \in S \}
Quotient Types

• A reinterpretation of quotient sets for Higher Order Logic
• Equivalence relations are represented by curried functions
  – \( E : \tau \rightarrow \tau \rightarrow \text{bool} \)
• Equivalence class of \( x \):
  \[ [x]_E = E x \]
  – Again it follows that \( [x]_E = [y]_E \iff E x y \)
• Define new quotient type \( \tau' \) as \( \tau / E \):
  – using HOL tool `define_new_type_bijections`
  – uses representing type \( \tau \rightarrow \text{bool}, \ldots \)
  – … restricted by predicate \( P \) where \( P f = \exists x. f = [x]_E \)
Quotient Maps and Quotient Theorems

- When a quotient type is defined, so are functions that map between the original and new types.
  - \( \llbracket \_ \rrbracket : \tau \rightarrow \tau' \)
  - \( \llbracket \_ \rrbracket : \tau' \rightarrow \tau \)

- These functions are automatically defined so that
  - \( \forall a. \llbracket \llbracket a \rrbracket \rrbracket = a \) \( a : \tau' \)
  - \( \forall r, r'. E r r' \iff (\llbracket r \rrbracket = \llbracket r' \rrbracket) \) \( r, r' : \tau \)

- The conjunction of these two properties occurs often and is called a “quotient theorem” for the type \( \tau' \)
New Quotient Type Definitions

• The main tool in the quotient package:

\[
\text{define_quotient_type} : \text{string} \to \text{string} \to \text{string} \to \\
\text{thm} \to \text{thm} \to \text{thm} \to \text{thm}
\]

\[
\text{define_quotient_type} \ "\text{tynanem}\" \ "\text{abs}\" \ "\text{rep}\"
\]

\[
\begin{align*}
\forall a: \text{ty}. \ & \text{EQUIV} \ a \ a \\
\forall a \ b: \text{ty}. \ & \text{EQUIV} \ a \ b = \text{EQUIV} \ b \ a \\
\forall a \ b \ c: \text{ty}. \ & \text{EQUIV} \ a \ b \ \land \ \text{EQUIV} \ b \ c \implies \text{EQUIV} \ a \ c
\end{align*}
\]

• defines the new type \text{tynanem} in the HOL logic
• defines the mapping functions \text{abs} and \text{rep}
• proves, stores, and yields the quotient theorem for \text{tynanem}
Quotient Theorem Consequences

• Here is a sample from a dozen properties automatically proven solely from a quotient theorem:

\[
\begin{align*}
\forall r. & \text{ EQUIV (rep (abs r)) r} \\
\forall a \ a'. & (a = a') = \text{ EQUIV (rep a) (rep a')} \\
\forall a \ a'. & (\text{rep a} = \text{rep a'}) = (a = a') \\
\forall r \ r'. & (\text{abs r} = \text{abs r'}) = (\text{EQUIV r} = \text{EQUIV r'}) \\
\forall r. & \exists a. \text{ EQUIV r (rep a)} \\
\forall a. & \exists r. \ a = \text{abs r} \\
\forall x \ y. & (\text{EQUIV x} = \text{EQUIV y}) = \text{EQUIV x y}
\end{align*}
\]
Aggregate Quotient Types

• At times one wishes to take the quotients of a family of related types, for example, a set of mutually recursive types.
• Could be done by multiple applications of define_quotient_type, BUT...
• If types $\tau_1$ and $\tau_2$ were related by $\tau_1 = (\tau_2)\text{list}$, then the quotient types $\tau_1'$ and $\tau_2'$ would NOT be related by $\tau_1' = (\tau_2')\text{list}$. (isomorphic, but not equal as HOL types)
• To avoid this, the quotient package provides several functions for constructing quotients of list types, pair types, and sum types. They do not construct new types, but do define new mapping functions.
Example: The Sigma Calculus

- Untyped sigma calculus
  - introduced by Abadi and Cardelli, A Theory of Objects
  - highlights the concept of objects, rather than functions
- Will first define an initial, “pre-” version
- Then create a refined, “pure” version using quotients
- Involves mutually recursive types
The Pre-Sigma Calculus

- Terms denote objects and methods. Sets of object terms $O_1$ and method terms $M_1$ are defined recursively by:

  - $x \in O_1$ for all variables $x$;
  - If $m_1, \ldots, m_n \in M_1$ then $[l_1=m_1, \ldots, l_n=m_n] \in O_1$ for all labels $l_1, \ldots, l_n$;
  - If $a \in O_1$ then $a.l \in O_1$ for all labels $l$;
  - If $a \in O_1$ and $m \in M_1$ then $a.l \leftarrow m \in O_1$ for all labels $l$;
  - If $a \in O_1$ then $\zeta(x)a \in M_1$ for all variables $x$. (binder)
The Pre-Sigma Calculus in HOL

• New nested mutually recursive types defined in HOL:

\[
\text{Val } _\_ = \text{Hol_datatype} \\
\quad \text{` obj1 = OVAR1 of var} \\
\quad \quad \quad \text{OBJ1 of (string # method1) list} \\
\quad \quad \quad \text{INVOKE1 of obj1 => string} \\
\quad \quad \quad \text{UPDATE1 of obj1 => string => method1;}
\]

\[
\text{method1 = SIGMA1 of var => obj1 ` ;}
\]

• Acts like the tool created not two but four new types:
  - entry1 = string # method1
  - dict1 = (entry1)list
The Pure Sigma Calculus in HOL

• Unfortunately we do not have $\xi(x)x = \xi(y)y$
• So we define the pure sigma calculus from the pre-sigma calculus in steps.
• The quotient theorems produced in some steps are used to perform later quotient operations.
• Assume that we have already defined or proven:
  – Alpha-equivalence relations for each of the four types: ALPHA_obj, ALPHA_dict, ALPHA_entry, ALPHA_method
  – Theorems proven for reflexivity, symmetry, transitivity: ALPHA_obj_REFL, ALPHA_obj_SYM, ALPHA_obj_TRANS, etc.
Defining the Pure Sigma Calculus Types

We define the pure sigma calculus types \texttt{obj} and \texttt{method}:

- \texttt{val obj\_QUOTIENT =}

  \texttt{define_quotient_type \text\texttt{"obj" \text\texttt{"obj\_ABS" \text\texttt{"obj\_REP"}}}
  \texttt{ALPHA\_obj\_REFL ALPHA\_obj\_SYM ALPHA\_obj\_TRANS;}

  \texttt{> val obj\_QUOTIENT =}
  \texttt{  \(\exists a. \text{obj\_ABS} (\text{obj\_REP} a) = a\) /
  \(\exists r r'. \text{ALPHA\_obj} r r' = (\text{obj\_ABS} r = \text{obj\_ABS} r')\))}

- \texttt{val method\_QUOTIENT =}

  \texttt{define_quotient_type \text\texttt{"method" \text\texttt{"method\_ABS" \text\texttt{"method\_REP"}}}
  \texttt{ALPHA\_method\_REFL ALPHA\_method\_SYM ALPHA\_method\_TRANS;}

  \texttt{> val method\_QUOTIENT =}
  \texttt{  \(\exists a. \text{method\_ABS} (\text{method\_REP} a) = a\) /
  \(\exists r r'. \text{ALPHA\_method} r r' = (\text{method\_ABS} r = \text{method\_ABS} r')\))}
Defining the Entry Mapping Functions

We define the abstraction and representation functions for the entry = (string # method) type, using the quotient theorem for methods, by:

- \[\begin{align*}
&\text{val (entry_MAPS_DEF, entry_QUOTIENT) =} \\
&\quad \text{new_pair_quotient_maps} \quad \text{“entry” “entry_ABS” “entry_REP”}
&\quad \text{ALPHA_entry_DEF TRUTH method_QUOTIENT;}
&
&\text{val entry_MAPS_DEF =}
&\quad \text{(!a b. entry_ABS (a,b) = (a, method_ABS b))} \\
&\quad \text{(!a b. entry_REP (a,b) = (a, method_REP b))}
&
&\text{val entry_QUOTIENT =}
&\quad \text{(!a. entry_ABS (entry_REP a) = a)} \\
&\quad \text{(!r r’. ALPHA_entry r r’ = (entry_ABS r = entry_ABS r’))}
\end{align*}\]
Defining the Dict Mapping Functions

Now we define the corresponding functions for the dict = (entry)list type, with the quotient theorem for entries:

```ml
- 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))
  val dict_QUOTIENT =
  |- (!a. dict_ABS (dict_REP a) = a) /
      (!r r'. ALPHA_dict r r' = (dict_ABS r = dict_ABS r'))
```
Defining Pure Constructor Functions

- We define constructor functions for the pure sigma calculus:

  - val OVAR_def = 
    Define `OVAR x = obj_ABS (OVAR1 x)`;

  - val OBJ_def = 
    Define `OBJ d = obj_ABS (OBJ1 (dict_REP d))`;

  - val SIGMA_def = 
    Define `SIGMA x a = method_ABS (SIGMA x (obj_REP a))`;

Now we have, as intended, $\zeta(x)x = \zeta(y)y$:

SIGMA x (OVAR x) = SIGMA y (OVAR y)
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.
- Some systems are convenient to model at one level of granularity, but have properties which are only true at another:
  - the pure sigma calculus is Church-Rosser, while the pre-sigma calculus is not.
- Available online:
  [http://www.cis.upenn.edu/~hol/lamcr](http://www.cis.upenn.edu/~hol/lamcr)