

# Higher Order Quotients and their Implementation in Isabelle HOL

Oscar Slotosch\*

Technische Universität München  
Institut für Informatik, 80290 München, Germany  
<http://www4.informatik.tu-muenchen.de/~slotosch/>  
[slotosch@informatik.tu-muenchen.de](mailto:slotosch@informatik.tu-muenchen.de)

**Abstract.** This paper describes the concept of higher order quotients and an implementation in Isabelle. Higher order quotients are a generalization of quotients. They use partial equivalence relations (PERs) instead of equivalence relations to group together different elements. This makes them applicable to arbitrary function spaces. Higher order quotients are conservatively implemented in the Isabelle logic HOL with a type constructor and a type class for PERs. Ordinary quotients are a special case of higher order quotients. An example shows how they can be used in Isabelle.

## 1 Introduction

Quotients are used in mathematics to group together different elements. This is done by defining an equivalence relation relating different elements. The quotient is a structure (type) consisting of groups (sets) of equivalent elements, called *equivalence classes*. Equivalent elements are in the same equivalence class. In formal system and software engineering quotients are used in many ways. For example to abstract from irrelevant details in the modelling of systems.

Due to the high complexity of systems and software, formal methods are used to support the correctness proof of realizations of systems with respect to the specification. Theorem provers are very useful in formal software development since they are tools to prove the correctness of the realization with respect to the specification. To prove abstract requirements we need theorem provers supporting quotients.

Functional languages are well suited for the development of systems, since they allow us to program in a clean and modular way. An important concept of functional languages is  $\lambda$ -abstraction which supports the formalization of abstract and higher order programs, which are highly reusable (a small example is the map functional). A higher order logic supports an adequate formalization of higher order programs.

---

\* This work is partially sponsored by the German Federal Ministry of Education and Research (BMBF) as part of the compound project “KorSys”.

Isabelle is a generic theorem prover. One logic for Isabelle is HOL, a higher order logic which goes back to [Chu40] and includes some extensions like polymorphism and type constructors of [GM93]. Isabelle HOL supports the definition of higher order functions, but a quotient construction is not yet available. In this work we define quotients and higher order quotients and give an implementation in Isabelle HOL.

This paper is structured as follows: First a short overview over the relevant concepts of Isabelle is given. It is described in Section 3 how quotients could be implemented in Isabelle HOL. Section 4 defines PERs (partial equivalence relations) and higher order quotients and describes an implementation of them in Isabelle HOL. This implementation includes the simple quotients as a special case. Theorems for higher order quotients are derived and compared to those for quotients. In Section 5 a method is presented for the definition of different quotients over the same type and an example is performed. Section 6 concludes with a summary, future work and comparisons to some related work.

## 2 Isabelle HOL

This section shortly presents the used concepts of Isabelle, especially of the logic HOL. For a more detailed description see [Pau94].

### 2.1 Defining Types

The logic HOL is strongly typed. Every type has to be non-empty<sup>2</sup>. The axiomatic declaration of types with the `types` section and some rules cannot ensure non-emptiness and therefore it can lead to inconsistent specifications<sup>3</sup>. Therefore we use the following definition form of Isabelle HOL to define types conservatively (like types in Gordon's HOL system [GM93]).

```
PNAT = Nat +
typedef pnat = "{p::nat.0<p}"    (PosNE)
end
```

In this example we introduce the type of positive natural numbers in a theory called `PNAT` which uses the theory `Nat`. The type of positive natural numbers (`pnat`) is defined to be (isomorphic to) the set of elements `p` of type `nat` (written `p::nat`) which fulfil the predicate `0<p`. The witness that the new type is not empty ( $\exists x. x \in \{p::nat. 0 < p\}$ ) is proved in a theorem over natural numbers before the type `pnat` is defined. This theorem is called `PosNE`. The `typedef` construct introduces the type only if the representing subset can be proved to be non-empty. With the given witness (`PosNE`) this proof is trivial.

---

<sup>2</sup> For example this is ensured by the `datatype` construct for the definition of free inductive data types (see [Völ95]).

<sup>3</sup> Inconsistent in the sense that it does not allow us to deduce `False`.

Isabelle HOL has no “real” subtyping, but subtypes may be introduced with coercion functions *abs* and *rep*. These coercion functions can be used to define functions on the subtype. For example `pnat_plus = λx y. Abs_pnat(Rep_pnat x + Rep_pnat y)`. The `typedef` construct in the example PNAT is equivalent to the following theory with explicit coercion functions:

```
PNAT = Nat + (* expanded typedef *)
consts (* signature of functions / constants *)
  pnat      :: "nat set"
  Abs_pnat  :: "nat set ⇒ pnat"
  Rep_pnat  :: "pnat ⇒ nat set"

defs (* definition of the subset *)
  pnat_def  "pnat ≡ {p. 0 < p}"

rules (* coercion rules *)
  Rep_pnat      "Rep_pnat x ∈ pnat"
  Abs_pnat_inverse "y ∈ pnat ⇒ Rep_pnat(Abs_pnat y) = y"
  Rep_pnat_inverse "Abs_pnat(Rep_pnat x) = x"

end
```

Axioms (names and rules) are declared in Isabelle after the keyword `rules`. Definitions are special rules (with the defining equality  $\equiv$ ). The keyword `defs` causes Isabelle to check whether a rule is a definition. Using only `defs` and `typedef` we cannot introduce inconsistencies.

In Isabelle HOL we may also use polymorphic types and type constructors. Type constructors can be seen as functions on types. For example `list` takes a type  $\alpha$  and maps it into the type  $\alpha$  `list`. Type constructors can also be defined using `typedef`.

## 2.2 Axiomatic Type Classes

Type classes are used to control polymorphism. For example in ML there are two main type classes ( $\alpha$  and  $\alpha_+$ ) to distinguish arguments of polymorphic functions that do not permit equality tests from those that do. As in other type systems (Haskell/Gofer [HJW92, Jon93]) Isabelle allows type classes to be defined with a subclass hierarchy.

In Isabelle there are different possibilities to introduce type classes<sup>4</sup>. We focus here on the defining form. This form is called *axiomatic type classes* in Isabelle (see [Wen94, Wen95, Wen97] for more details).

Axiomatic type classes characterize a type class by some axioms. Instantiating a type into a type class requires to prove that these axioms hold for the type. We explain the axiomatic type classes on the example of equivalence relations.

---

<sup>4</sup> Type classes are also called “sorts”.

```

ER = HOL + (* theory of equivalence relations *)

consts (* polymorphic (infix) constant *)
  "~~"   :: "α::term ⇒ α ⇒ bool" (infixl 55)

axclass er < term
  (* axioms for equivalence relations *)
  ax_er_refl    "x ~~ x"
  ax_er_sym     "x ~~ y ⇒ y ~~ x"
  ax_er_trans   "[x ~~ y; y ~~ z] ⇒ x ~~ z"

```

To characterize the type class `er` of equivalence relations we introduced a constant `~~`, available on all types of the class `term`<sup>5</sup>. A type belongs to the class `er`, if `~~` on that type satisfies the axioms `ax_er_refl`, `ax_er_sym` and `ax_er_trans`. To show the instantiation of a type into a type class we define now a theory to instantiate the type `pnat` into the class `er` (with the total equivalence relation `True`).

```

PNAT2 = PNAT + ER +
defs   (* concrete definition of ~~ on pnat *)
  er_pnat_def " (op ~~) ≡ (λx y::pnat.True)"
end

```

For this theory we can trivially derive the following theorems:

```

er_refl_pnat  "(x::pnat) ~~ x"
er_sym_pnat   "(x::pnat) ~~ y ⇒ y ~~ x"
er_trans_pnat "[ (x::pnat) ~~ y; y ~~ z ] ⇒ x ~~ z"

```

These theorems ensure that the type `pnat` belongs to the class `er`. The instantiation of `pnat` needs these theorems as witnesses:

```

PNAT3 = PNAT2 + (* instance of pnat into er *)
instance pnat::er (er_refl_pnat,er_sym_pnat,er_trans_pnat)
end

```

This instantiation makes all theorems for `α::er` applicable to `pnat`.

Using axiomatic type classes is a conservative form to introduce type classes and instances into the specifications. In contrast to the introduction of type classes with the constructs `classes` and `arities` (see for example [Reg94]) axiomatic type classes do not require to provide an instance as witness that the type class is not empty<sup>6</sup>. However using axiomatic type classes in this way has a small disadvantage: type checking cannot ensure that equivalence relations are only applied to terms of types belonging to the class `er`. For example we are

<sup>5</sup> `term` is the universal class in Isabelle HOL to which all HOL types automatically belong.

<sup>6</sup> In [Reg94] the correctness of instantiations is justified by extra-logical arguments and therefore it is not checked by Isabelle.

allowed to write the term  $(n::\text{nat}) \sim\sim n$ , even if we fail to prove it, if `nat` is not instantiated into `er`.

Therefore we introduce an additional constant into the specification `ER`, which is only available on types of class `er`.

```
(* ER (continued) *)
consts (* characteristic constant for er *)
      "~"      :: "α::er ⇒ α ⇒ bool"   (infixl 55)
defs
  er_def      "(op ~) ≡ (op ~~)"
end
```

We call those constants *characteristic constants* and we can easily derive the *characteristic axioms* for it.

```
er_refl      "x ~ x"
er_sym       "x ~ y ⇒ y ~ x"
er_trans     "[x ~ y; y ~ z] ⇒ x ~ z"
```

We can use  $\sim$  in all general proofs about equivalence relations. The constant  $\sim\sim$  is only used for the correct instantiation of types. To expand the definition of  $\sim$  on `pnat` we derive the following *instantiation rule*:

```
inst_pnat_er "(op ~) = (λx y::pnat.True)
```

With this rule we do not need the overloaded constant  $\sim\sim$  on `pnat` any more.

Using  $\sim$  instead of  $\sim\sim$  has the advantage that the type checker can prohibit us from writing strange terms (like  $n::\text{nat}\sim\sim n$  if on `nat` no concrete definition of  $\sim\sim$  is given). Trying to prove  $(n::\text{nat}) \sim n$  would result in a type error, if `nat` is not instantiated into the class `er`.

Since HOL has also type constructors for higher order types, the instantiation rule allows us to describe the class of the result of a type constructor, provided the arguments of the type constructor are of certain classes<sup>7</sup>. The statement:

```
instance list::(er)er   (er_refl_list,er_sym_list,er_trans_list)
```

Instantiates all types  $\alpha::\text{er}$  `list` into the class `er`<sup>8</sup>. In other words: If we have a concrete type with a concrete definition of the equivalence relation  $\sim\sim$ , then we also have a concrete definition of equivalence relation on the lists over this type. Type constructors and type definitions are illustrated on the example in the following section with more details.

<sup>7</sup> With the same restrictions as the `arity` declaration of Isabelle (see [Nip91] for more details on Isabelle's type system).

<sup>8</sup> We omitted the definition of the equivalence  $\sim\sim$  on lists and the witnesses here for brevity.

### 3 Quotients in Isabelle HOL

This section shows how type definitions, type constructors and type classes can be used to define a quotient constructor in Isabelle. The used techniques are the same as in Section 4 for higher order quotients.

First we recall the definitions of equivalence relations and quotients:

**Definition 1.** Equivalence Relation

A relation  $\sim$  on a type  $R$  ( $\sim \subseteq R \times R$ ) is called *equivalence relation*, iff

- REFLEXIVE:  $\forall x \in R. x \sim x$
- SYMMETRIC:  $\forall x, y \in R. x \sim y$  implies  $y \sim x$
- TRANSITIVE:  $\forall x, y, z \in R. x \sim y$  and  $y \sim z$  implies  $x \sim z$

Equivalence relations are formalized in Isabelle HOL in the theory ER in Section 2.2.

**Definition 2.** Quotient

Let  $\sim$  be an equivalence relation on  $S$ . Then the *quotient* (of  $S$  with respect to  $\sim$ ) is the set of all *equivalence classes*, defined by:

- QUOTIENT:  $S/\sim := \{[x]_\sim \mid x \in S\}$  where
- EQUIVALENCE CLASS:  $[x]_\sim := \{y \in S \mid x \sim y\}$  for all  $x \in S$

To implement quotients in Isabelle we define a type constructor `quot` which is defined on elements of the class `er`. This is done in the following theory:

```
QUOT = ER +
typedef  $\alpha$  quot = "{s.  $\exists r. \forall y. y \in s \Rightarrow y \sim r$ }" (QuotNE)
```

With this `typedef` the quotient type constructor `quot` is defined. The representing set of elements is defined as set of equivalence classes. It contains such elements `s` for which a (representative) element `r` exists, such that all elements in `s` are equivalent to this element `r`. The theorem `QuotNE` states that this set is not empty. The non-emptiness does not depend on properties of  $\sim$ .

As was mentioned in Section 2.1 the `typedef` construct introduces abstraction and representation functions (`Abs_quot` and `Rep_quot`), for quotients. With these functions we can define the equivalence class operator and the operator which picks a single element out of an equivalence class by<sup>9</sup>:

```
(* QUOT (continued) *)
consts
  eclass      :: " $\alpha :: er \Rightarrow \alpha$  quot"
  any_in     :: " $\alpha :: er$  quot  $\Rightarrow \alpha$ "
defs
  eclass_def  " $\langle [x] \rangle \equiv \text{Abs\_quot } \{y. y \sim x\}$ "
  any_in_def  " $\text{any\_in } f \equiv @x. \langle [x] \rangle = f$ "
end
```

<sup>9</sup> For readability we omit here the definitions which cause Isabelle to use `<[ ]>` as infix syntax.

For these operators we can derive a lot of useful theorems<sup>10</sup>, which make it easy to use quotients in Isabelle HOL. The types of  $x$  and  $y$  are inferred automatically to be  $\alpha :: \text{er}$ .

```
(* theorems for equality *)
er_class_eqI  "x~y==><[x]>=<[y]>"
er_class_eqE  "<[x]>=<[y]>==>x~y"
er_class_eq   "<[x]>=<[y]>=x~y"

(* theorems for inequality *)
er_class_neqI "¬x~y==><[x]>≠<[y]>"
er_class_neqE "<[x]>≠<[y]>==>¬x~y"
er_class_neq  "<[x]>≠<[y]>=(¬x~y)"

(* theorems for exhaustiveness and induction *)
er_class_exh  "∃x.q=<[x]>"
er_class_all  "∀x.P<[x]> ==> P q"

(* theorems for any_in *)
any_in_class  "any_in <[x]> ~ x"
class_any_in  "<[any_in q]> = q"
```

The theorem `er_class_eqI` states that equivalent elements are in the same equivalence class. With the quotient construction, the defined operations, and these theorems we can lift theorems and operations from the representations to the quotients easily. Using quotient types in Isabelle is now as easy as in mathematics:

1. Define a relation on the representing type.
2. Prove that it is an equivalence relation and instantiate the representing type into the class `er`.
3. Build the quotient type with the quotient constructor
4. Define functions on the quotient with `< [ ]>` and `any_in`.

A more detailed example is the construction of rational numbers in Section 5.

## 4 Higher Order Quotients in Isabelle

This section defines higher order quotients, a generalization of quotients, and gives an implementation in Isabelle HOL. Higher order quotients are called higher order, since they can be applied to higher order functions, without explicitly defining an (partial) equivalence relation for this higher order types.

Quotients can be used on every type which belongs to the class `er`. Therefore using quotients requires to have an equivalence relation on every representing type. At the end of Section 2.2 we mentioned the possibility to define an equivalence relation for lists, based on an equivalence relation of the elements. This

---

<sup>10</sup> We present them here to show how quotients may be used. We will not apply all of them in the rest of this paper.

instance is quite nice, since it allows us to build quotients over specific lists, without explicitly defining the equivalence relation (for example with an equivalence relation on `pnat` we can define types like `types PLQ = pnat list quot`).

In functional languages we have  $\lambda$ -abstraction and we can program functions of arbitrary higher order types. However we cannot build quotients over these functions unless we defined an equivalence relation for every function type. The reason is that, in contrast to lists, we cannot define an equivalence relation for arbitrary functions, because in general equivalence relations on functions are not reflexive (since  $x \sim y \not\Rightarrow f(x) \sim f(y)$ ).

Higher order quotients use partial equivalence relations (PERs) instead of equivalence relations to group together different elements. For PERs we can give a general PER on functions, which defines a PER on the function space, provided that domain and range of the functions belong to the class PER. It was already observed in [Rob89] that PERs are closed under functional composition. With higher order quotients we can also define types like `types FQ = (pnat list  $\Rightarrow$  pnat) quot`.

#### 4.1 PERs

Partial equivalence relations (PERs) are the basis for higher order quotients. PERs are not necessarily reflexive. PERs have a domain on which they are reflexive. Therefore they are called *partial* equivalence relations.

##### Definition 3. Partial Equivalence Relation

A relation  $\sim$  on a type  $R$  ( $\sim \subseteq R \times R$ ) is called *partial equivalence relation*, iff

- SYMMETRIC:  $\forall x, y \in R. x \sim y$  implies  $y \sim x$
- TRANSITIVE:  $\forall x, y, z \in R. x \sim y$  and  $y \sim z$  implies  $x \sim z$

The domain  $D$  of a PER is the set of values from  $R$ , on which  $\sim$  is reflexive:

- $D := \{x \in R. x \sim x\}$

PERs are called partial equivalence relations, since they are, in contrast to equivalence relations, reflexive only on the domain  $D$ .

From these axioms we can derive (by symmetry and transitivity) that all values not in the domain  $D$  are not partially equivalent.

- $x \sim y$  implies  $x, y \in D$
- $x \notin D$  implies  $x \not\sim y$

The formalization of PERs in Isabelle HOL is similar to the formalization of equivalence relations in Section 2.2. It uses a polymorphic definition and an axiomatic type class. This definition includes equivalence relations as a special case. This is expressed with the subclass relation. The advantage of this hierarchy is that all theorems derived for types with partial equivalence relations are also available on types with equivalence relations. Furthermore we can derive all theorems of Section 3 by restricting the polymorphism to the types which have an equivalence relation defined.



```

PERO = Set + (* PERs and ER *)

consts (* polymorphic constant *)
  "~~"    :: "α::term ⇒ α ⇒ bool" (infixl 55)

axclass per < term (* PERs *)
  ax_per_sym      "x ~~ y ⇒ y ~~ x"
  ax_per_trans    "[x ~~ y; y ~~ z] ⇒ x ~~ z"

axclass er < per (* ERs are a subclass of PERs *)
  ax_er_refl      "x ~~ x"

consts (* characteristic constant and Domain for per *)
  "~"           :: "α::per ⇒ α ⇒ bool" (infixl 55)
  D             :: "α::per set"

defs
  per_def      "(op ~) ≡ (op ~~)"
  Domain      "D ≡ {x.x~x}"
(* define ~~ on function type ⇒ *)
  fun_per_def  "f~~g ≡ ∀x y.x∈D∧y∈D∧x~y → f x~g y"
end

```

This theory contains the definition of a PER on the function space. The following theorems can be derived for PERs:

```

(* characteristic axioms for ~ *)
  per_sym      "x ~ y ⇒ y ~ x"
  per_trans    "[x ~ y; y ~ z] ⇒ x ~ z"
(* some theorems for ~ and the Domain D *)
  sym2refl1    "x ~ y ⇒ x ~ x"
  sym2refl2    "x ~ y ⇒ y ~ y"
  DomainD      "x ∈ D ⇒ x ~ x"
  DomainI      "x ~ x ⇒ x ∈ D"
  DomainEq     "x ∈ D = x ~ x"
  DomainI_left "x ~ y ⇒ x ∈ D"
  DomainI_right "x ~ y ⇒ y ∈ D"
  notDomainE1  "x ∉ D ⇒ ¬ x ~ y"
  notDomainE2  "y ∉ D ⇒ ¬ x ~ y"
(* theorems for equivalence relations *)
  er_refl      "(x::α::er) ~ x"
  er_Domain    "(x::α::er) ∈ D"
(* witnesses for "⇒" ::(per,per)per *)
  per_sym_fun  "(f::α::per ⇒ β::per)~~g ⇒ g~~f"
  per_trans_fun "[f::α::per⇒β::per)~~g;g~~h]⇒f~~h"

```

The last two theorems allow us to instantiate the function space into the class `per` in the following theory:

```

PER = PER0 + (* instance for per *)

instance fun :: (per,per)per (per_sym_fun,per_trans_fun)

end

```

To expand the PER on functions without using  $\sim\sim$  we deduce the instantiation rule:

```

inst_fun_per "f~g=( $\forall x y. x \in D \wedge y \in D \wedge x \sim y \longrightarrow f x \sim g y$ )"

```

## 4.2 Higher Order Quotients

This section defines higher order quotients and gives an implementation in Isabelle HOL which generalizes the quotient implementation. The goal is that we can derive similar theorems for higher order quotients as for quotients and to apply them to quotients as a special case<sup>11</sup>.

**Definition 4.** Higher Order Quotient Let  $\sim$  be an partial equivalence relation on  $S$ . Then the *higher order quotient* is the set of all *partial equivalence classes*, defined by:

- HIGHER ORDER QUOTIENT:  $S/\sim := \{[x]_\sim \mid x \in S\}$  where
- PARTIAL EQUIVALENCE CLASS:  $[x]_\sim := \{y \in S \mid x \sim y\}$  for all  $x \in S$

In contrast to the definition of equivalence classes, partial equivalence classes may be empty sets. To implement higher quotients in Isabelle we define a type constructor `quot` which is defined on elements of the class `per`. This is done in the following theory:

```

HQUT = PER +
typedef  $\alpha$  quot = "{s.  $\exists r. \forall y. y \in s \Rightarrow y \sim r$ }" (QuotNE)

```

With this `typedef` the quotient type constructor `quot` is defined. This type is not empty, even if the PER is always false. In this case the type contains the empty set as only element.

As in the example of quotients we define the partial equivalence class operator and the function `any_in` by:

```

(* HQUT (continued) *)
consts
  peclass      :: " $\alpha :: \text{per} \Rightarrow \alpha \text{ quot}$ "
  any_in       :: " $\alpha :: \text{per} \text{ quot} \Rightarrow \alpha$ "
defs
  peclass_def  "class <[x]>  $\equiv$  Abs_quot {y. y ~ x}"
  any_in_def   "any_in f  $\equiv$  @x. <[x]> = f"
end

```

<sup>11</sup> Of course we can derive all theorems for quotients also for reflexive (higher order) quotients.

Of course we can derive all theorems for quotients (see Section 3) also for reflexive (higher order) quotients. The only difference is that we have to use the class `er` for the type variables. For example:

```
er_class_eq <[(x::α::er)]>=<[y]>=x~y
```

Some theorems for higher order quotients are like those for quotients, but others need an additional premise, since the concept is more general.

```
(* theorems for equality *)
per_class_eqI "x~y ==> <[x]>=<[y]>"
per_class_eqE "[x∈D; <[x]>=<[y]>] ==> x~y"
per_class_eq "x∈D ==> <[x]>=<[y]>=x~y"

(* theorems for inequality *)
per_class_neqI "[x∈D; ¬x~y] ==> <[x]>≠<[y]>"
per_class_neqE "<[x]>≠<[y]> ==> ¬x~y"
per_class_neq "x∈D ==> <[x]>≠<[y]>=(¬x~y)"

(* theorems for exhaustiveness and induction *)
per_class_exh "∃x.q=<[x]>"
per_class_all "∀x.P<[x]> ==> P q"

(* theorems for any_in *)
per_any_in_class "x∈D ==> any_in <[x]> ~ x"
per_class_any_in "∀x::α::per.x∈D ==>
  <[any_in(q::α::per quot)]>=q"
```

The additional premise `x∈D` is required, since there may be different elements which are not in relation to each other. All those elements would be represented by the empty partial equivalence class. Therefore we cannot deduce from the equality of equivalence classes that the elements are equivalent. This problem can be solved, if we define the PER such that it is reflexive for all elements, except for one element. Using  $\perp$  as the only element which is not reflexive integrates our concept of PERs with the Scott-domains (see [Slo97] for more details).

With these polymorphic theorems we can transform theorems from representations to quotients without schematically proving them again. The reason for this is that the axiomatic type classes allow us to tailor the polymorphism to our constructions. With axiomatic type classes we can justify the use of type classes without extra-logical arguments and they allow us to prove all necessary theorems within the system.

Comparing our construction with the schematic construction of quotients shows us, beside the fact that we use PERs and higher order quotients, the argument, that we can instantiate a type only once into a type class and we can therefore have at most one quotient over every type. This drawback is overcome by an embedding method, presented in the following section. The method allows us to *reuse* a basic type for multiple quotient definitions by schematically constructing another type for the definition of the PER<sup>12</sup>.

<sup>12</sup> Note that we do not aim at the construction of value dependent (quotient) types, like the integers modulo  $n$  which would exceed Isabelle's type system. Therefore the presented method suffices for "static types".

## 5 Example: Rational Numbers

This section presents the application of (higher order) quotients on the fractional representation of rational numbers. The intended quotient is to identify terms like  $\frac{1}{2}$  with  $\frac{2}{4}$ . This example could also be treated with ordinary quotients, however we selected it, since it is small, well-known, and suffices to illustrate the method for using quotients. The emphasis lies not only in the simple task of dealing with fractions, but also in the embedding method which allows us to reuse a type as basis for several quotients. Higher order quotients can be applied in the same way, for example to identify observational equivalent functions.

Our quotient constructor builds the quotient with respect to a fixed (partial) equivalence relation. This relation is fixed for each type by the instance of this type into the class `per`. Encoding the PER into the type class has the advantage, that the operation `<[ ]>` does not need an additional parameter for the PER and that we may have a general type constructor for the construction of quotient types. The drawback of this encoding is that on every type we have at most one PER.

In our example we represent rational numbers by pairs of natural numbers and positive natural numbers. However, if we have PERs on positive and natural numbers, and a general PER for pairs ( $p \sim\sim q = (\text{fst } p \sim \text{fst } q \wedge \text{snd } p \sim \text{snd } q)$ ), then we have already a PER defined on pairs of natural numbers. We could build quotients over it, but this is not the PER we need to identify terms like  $\frac{1}{2}$  with  $\frac{2}{4}$ . In addition the Isabelle type system does not allow us to instance the type `(nat * pnat)` directly into the class `per`, since it permits only some special forms of arities<sup>13</sup>.

To avoid these problems, we need an embedding. We define a new representing type with the `typedef` construct.

```
NPAIR = Arith + Prod + PER + PNAT +
(* embedded representation: *)

typedef NP = "{n::nat*pnat.True}"

end
```

With this representing type we define the equivalence relation by:

```
(* NPAIR (continued) *)
defs   per_NP_def " (op ~\~)  $\equiv$ 
          ( $\lambda x y.$ fst(Rep_NP x) * Rep_pnat(snd(Rep_NP y)) =
            fst(Rep_NP y) * Rep_pnat(snd(Rep_NP x))) "
```

This definition might look quite technical, but with the theorems from the `typedef` construction with the predicate `True` the embedding functions for NP are isomorphisms and therefore they disappear in the proofs using the simplifier.

---

<sup>13</sup> Arities are a very restricted form of schematic properties about type constructors.

Since we have no special multiplication for natural numbers with positive numbers, we use the representation function `Rep_pnat`, defined in Section 2.1. Eliminating this representation function with the corresponding abstraction function `Abs_pnat` requires that the argument is a positive natural number (see the following proof).

The next step is to prove that this relation is a PER. We prove:

```
per_sym_NP    "(x::NP) ~ y ==> y ~ x"
per_trans_NP "[[x::NP] ~ y; y ~ z] ==> x ~ z"
```

We do not need to prove reflexivity, since we are using the higher order quotient construct, which requires only a PER. To prove transitivity of this equivalence relation was the only non-trivial task in this example.

After the instantiation of our representation into the class `per` we define the type `fract` as quotient of our representation.

```
FRACT = NPAIR + HQUOT +
instance NP::per      (per_sym_NP,per_trans_NP)
  (* now define fractions *)
types  fract = NP quot
  (* example for fractions *)
consts half      :: "fract"
defs    half_def  "half ≡ <[Abs_NP(1,Abs_pnat 2)]>"
end
```

We derive the instantiation rule for the representation:

```
inst_NP_per "(op ~)=(λx y.fst(Rep_NP x)*Rep_pnat(snd(Rep_NP y))=
fst(Rep_NP y)*Rep_pnat(snd(Rep_NP x)))"
```

As an example for an application consider the proof of the following theorem:

```
> val prems = goalw thy [half_def]
  "0 < n ==> half = <[ Abs_NP(n,Abs_pnat(2*n)) ]>";
> by (cut_facts_tac prems 1);
```

```
Level 1
half = <[ abs_NP (n, 2 * n) ]>
1. 0 < n ==>
  <[ Abs_NP (1, Abs_pnat 2) ]> =
  <[ Abs_NP (n, Abs_pnat (2 * n)) ]>
```

```
(* derive that 0 < n+n (=2*n) *)
> by (dres_inst_tac [("m","n")] trans_less_add2 1);
```

```
Level 2
half = <[ Abs_NP (n, Abs_pnat (2 * n)) ]>
1. 0 < n + n ==>
```

```

    <[ Abs_NP (1, Abs_pnat 2) ]> =
    <[ Abs_NP (n, Abs_pnat (2 * n)) ]>

> fr per_class_eqI;

Subgoal 1 selected
Level 3
half = <[ Abs_NP (n, Abs_pnat (2 * n)) ]>
  1. 0 < n + n  $\implies$ 
      Abs_NP (1, Abs_pnat 2)  $\sim$ 
      Abs_NP (n, Abs_pnat (2 * n))

> by (simp_tac (!simpset addsimps [inst_NP_per]) 1);
Level 4
half = <[ Abs_NP (n, Abs_pnat (2 * n)) ]>
No subgoals!

```

This proof shows how the equality between quotients is reduced to the equivalence of the representations and that the embedding functions for the PER do not harm. The embedding of positive natural numbers requires a proof step. With the instantiation rule and some arithmetic knowledge the proof is completed automatically.

This example shows how easy quotients can be applied, even if an embedding is used for the representation.

## 6 Conclusion

In this work we used axiomatic type classes and type constructors to build a quotient construction for Isabelle HOL. Using PERs instead of equivalence relations allows us to build quotients over arbitrary functions and saves us from proving reflexivity. Quotients can be used as a special case of higher order quotients. Axiomatic type classes are applied to tailor the use of polymorphism to a polymorphic construction which requires neither schematic proofs nor any extra-logical justification.

Because partial equivalence classes can be empty, some theorems for higher order quotients are not as general as those for quotients. However a combination with Scott-domains provides a solution [Slo97]. The problem that type classes do not allow us to have multiple PERs on one type is methodically avoided with the embeddings presented in the example.

PERs are a very general concept in theoretical computer science. They are used for example as models in type theory [BM92]. In the field of algebraic specifications PERs are used for the implementation of ADTs. The implementation of ADTs consists of a restriction to a subtype, followed by a quotient step. Both can be expressed by PERs (for a formalization of implementation of ADTs in first order logic with PERs see for example [BH95]). PERs can also be used to

define a predicate to characterize observer functions and observability in a higher order logic (see [Slo97]). There has also been some work on quotients in theorem provers (for example in [Har96], or some work from T. Kalker presented at the 1990 HOL workshop without proceedings), but these approaches do not use the advantages of polymorphism in combination with the axiomatic type classes and they cannot build quotients over arbitrary function spaces.

An interesting application of quotients is to formalize the concept of states. Consider a functional description of a software module. From an observational point of view the module receives a stream of inputs and answers with a stream of outputs. The variables in the specification and realization of this module can be expressed as quotients over input histories. If the quotient over these possibly infinite input streams is finite, model checking can be used to prove the correctness of the module. So quotients can combine functional system descriptions with states and model checking. The integration of quotients into a functional development method is described in [Slo97], the combination with model checking is ongoing work in our research project KORSYS.

With this work we integrated PERs and quotients into Isabelle HOL and can use Isabelle now for formal development with quotients over arbitrary types.

**Acknowledgments:** I thank Tobias Nipkow for discussions on quotients and their implementation in Isabelle. For many comments on draft versions of this paper I thank Markus Wenzel and Jan Philipps. Furthermore I thank the anonymous referees for their constructive comments.

## References

- [BH95] Michel Bidoit and Rolf Hennicker. Behavioural Theories and The Proof of Behavioural Properties. Technical report, Paris, 1995.
- [BM92] Kim Bruce and John C. Mitchell. PER models of subtyping, recursive types and higher-order polymorphism. In *Principles of Programming Languages 19*, pages 316–327, Albuquerque, New Mexico, 1992.
- [Chu40] Alonzo Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5:56–68, 1940.
- [GM93] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
- [Har96] John Robert Harrison. *Theorem Proving with the Real Numbers*. PhD thesis, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1996. Technical Report No 408.
- [HJW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)*. ACM SIGPLAN Notices, May 1992.
- [Jon93] M. P. Jones. *An Introduction to Gofer*, August 1993.
- [Nip91] T. Nipkow. Order-Sorted Polymorphism in Isabelle. In G. Huet, G. Plotkin, and C. Jones, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321, 1991.
- [Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.

- [Reg94] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, 1994.
- [Rob89] Edmund Robinson. How Complete is PER? In *Fourth Annual Symposium on Logic in Computer Science*, pages 106–111, 1989.
- [Slo97] Oscar Slotosch. *Refinements in HOLCF: Implementation of Interactive Systems*. PhD thesis, Technische Universität München, 1997.
- [Völ95] Norbert Völker. On the Representation of Datatypes in Isabelle/HOL. Technical Report 379, University of Cambridge Computer Laboratory, 1995. Proceedings of the First Isabelle Users Workshop.
- [Wen94] Markus Wenzel. Axiomatische Typklassen in Isabelle. Master's thesis, Institut für Informatik, TU München, 1994.
- [Wen95] M. Wenzel.  
*Using axiomatic type classes in Isabelle — a tutorial*, 1995. Available at <http://www4.informatik.tu-muenchen.de/~nipkow/isadist/axclass.dvi.gz>.
- [Wen97] M. Wenzel. Type Classes and Overloading in Higher-Order Logic. In *Proceedings of Theorem Proving in Higher Order Logics*, 1997. in this volume.