# Defining Functions on Equivalence Classes

LAWRENCE C. PAULSON
University of Cambridge

A *quotient construction* defines an abstract type from a concrete type, using an equivalence relation to identify elements of the concrete type that are to be regarded as indistinguishable. The elements of a quotient type are *equivalence classes*: sets of equivalent concrete values. Simple techniques are presented for defining and reasoning about quotient constructions, based on a general lemma library concerning functions that operate on equivalence classes. The techniques are applied to a definition of the integers from the natural numbers, and then to the definition of a recursive datatype satisfying equational constraints.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic**]: Mechanical theorem proving; G2.0 [**Discrete Mathematics**]: General

General Terms: Theory; Verification

Additional Key Words and Phrases: equivalence classes, quotients, theorem proving

## 1. INTRODUCTION

Equivalence classes and quotient constructions are familiar to every student of discrete mathematics. They are frequently used for defining abstract objects. A typical example from the $\lambda$-calculus is $\alpha$-equivalence, which identifies terms that differ only by the names of bound variables. Strictly speaking, we represent a term by the set of all terms that can be obtained from it by renaming variables. In order to define a function on terms, we must show that the result is independent of the choice of variable names. Here we see the drawbacks of using equivalence classes: individuals are represented by sets and function definitions carry a proof obligation.

Users of theorem proving tools are naturally inclined to prefer concrete methods whenever possible. With their backgrounds in computer science, they will see functions as algorithms and seek to find canonical representations of elements. We can replace $\alpha$-equivalence by de Bruijn variables [de Bruijn 1972]. Other common applications of equivalence classes can similarly be replaced by clever data structures. However, such methods can make proofs unnecessarily difficult.

The automated reasoning community already uses equivalence classes. I conducted an informal e-mail survey and learned of applications using ACL2, Coq, HOL, Mizar and PVS. There must be other applications. However, ACL2 [Kaufmann and Moore 2002] does not use equivalence classes; it obtains a similar effect

*Author's address*: Computer Laboratory, 15 JJ Thomson Avenue, Cambridge CB3 0FD, England.
*E-mail*: lcp@cl.cam.ac.uk

through its ability to rewrite a term with respect to any equivalence relation.[1] The work in Coq is specific to constructive type theory and uses the concept of setoid [Geuvers et al. 2002]. The work using HOL requires the use of special purpose code [Harrison 1994; Homeier 2001]. If we consider the huge amount of mathematics that has been formalised, it seems that uses of equivalence classes are rare.

The object of this paper is to show that with the right definitions, working with equivalence classes is easy. The approach will work in any theorem prover that admits the formalisation of basic set theory. Function definitions are made in a stylised but readable form. Once a function is shown to respect the equivalence relation, further proofs can be undertaken as they would be in a mathematics textbook: by assuming as given an arbitrary representative of the equivalence class. Higher-order logic is sufficient, by the obvious representation of typed sets by predicates; untyped axiomatic set theory is also sufficient. The axiom of choice is not required. The tool does not need to be programmable. The examples in this paper have been done using Isabelle/HOL [Nipkow et al. 2002].

The paper provides a brief review of equivalence classes (§2). It then outlines a formal lemma library for equivalence classes (§3), which it demonstrates using a construction of the integers (§4). It then demonstrates how equational constraints can be imposed on a recursive datatype (§5). Finally, the paper presents brief conclusions (§6).

## 2. EQUIVALENCE CLASSES AND QUOTIENT SETS

This section is a brief review of the mathematics required for this paper. Thorough descriptions of the concepts appear in standard textbooks [Kolman et al. 2000, p. 128].

An *equivalence relation* over a set is any relation that is reflexive, symmetric and transitive. Each of the following examples is easily shown to be an equivalence relation.

—$\alpha$-equivalence has already been mentioned in the introduction.

—The integers can be defined as equivalence classes on pairs of natural numbers related by $(x, y) \sim (u, v)$ if and only if $x + v = u + y$.

—The rational numbers can be defined as equivalence classes on pairs of integers related by $(x, y) \sim (u, v)$ if and only if $xv = uy$, where $y,\ v \neq 0$. Since $y$ and $v$ must be non-zero, it is an equivalence relation on the set $\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$.

We can often do without equivalence classes. The integers can be defined using a signed magnitude representation. The rational numbers can be defined as fractions in reduced form. The drawbacks of both representations become clear when we try to prove that addition is associative. A signed magnitude representation requires case analysis on the sign and the consideration of eight cases; proving the associative law for reduced fractions will require a body of theory about greatest common divisors [Harrison 1994, §3].

If $\sim$ is an equivalence relation over a set $A$, and $x \in A$, then $[x]_\sim$ denotes $\{y \mid y \sim x\}$ and is called an *equivalence class*. The notation simplifies to $[x]$ when

[1] Documentation is on the Internet at `http://www.cs.utexas.edu/users/moore/acl2/v2-7/EQUIVALENCE.html`

$\sim$ is fixed in advance. Because $\sim$ is reflexive, an equivalence class is always non-empty. Because $\sim$ is symmetric and transitive, equivalence classes are disjoint: if two equivalence classes have an element in common, then they are equal. The equivalence relation therefore partitions the set $A$. The quotient set $A/\sim$ is defined to be the set of equivalence classes generated by $\sim$.

Let us examine the construction of the integers. Here $A$ is the set of pairs of natural numbers. As mentioned above, the equivalence relation is $(x, y) \sim (u, v)$ if and only if $x + v = u + y$. The integer 0 and functions for negation, addition and multiplication are defined in terms of equivalence classes.

$$\begin{aligned} 0 &= [(0,0)] \\ -\,[(x,y)] &= [(y,x)] \\ [(x,y)] + [(u,v)] &= [(x+u, y+v)] \\ [(x,y)] \times [(u,v)] &= [(xu+yv, xv+vu)] \end{aligned}$$

The intuition is that $[(x, y)]$ represents the integer $x - y$, so clearly the negation of $[(x, y)]$ must be $[(y, x)]$. The definition of multiplication is obtained by evaluating the product $(x - y)(u - v)$.

Such definitions are only legitimate if they are independent of the particular elements chosen from the equivalence classes. For example, negation is well-defined because $[(x, y)] = [(x', y')]$ implies $-[(x, y)] = -[(x', y')]$. The statement that a function respects an equivalence relation is called a *congruence property*. For negation, it is easily verified: if $(x, y) \sim (x', y')$, then $x+y' = x'+y$, so $y+x' = y'+x$ and thus $(y, x) \sim (y', x')$. The congruence property for addition is harder to verify and that for multiplication is harder still. These proof obligations are the chief drawback of using equivalence classes. Once we have verified the congruence properties, developing the theory of the integers is easy.

To prove $-(-z) = z$, write the integer $z$ as $[(x, y)]$. (We have just proved that the choice of $x$ and $y$ is irrelevant.) Now trivially

$$-(-[(x,y)]) = -[(y,x)] = [(x,y)].$$

The proof that addition is associative appeals to the corresponding property for the natural numbers:

$$\begin{aligned} \big([(x_1,y_1)] + [(x_2,y_2)]\big) + [(x_3,y_3)] &= [(x_1+x_2+x_3, y_1+y_2+y_3])] \\ &= [(x_1,y_1)] + \big([(x_2,y_2)] + [(x_3,y_3)]\big). \end{aligned}$$

All of these proofs work in the same way. We write the given integers as pairs of natural numbers. We simplify the expression using function definitions that have already been shown to be legitimate. We are left with elementary reasoning about the natural numbers. If we formalise equivalence relations appropriately, the formal proofs will be as natural as those shown above.

## 3. A FORMALIZATION OF EQUIVALENCE CLASSES

The key to effective use of equivalence classes is to give definitions in a particular form and to simplify them using particular lemmas. The formalization is designed to make the machine proofs as simple as possible, for any verifier. Unions capture

the possible ambiguity in a function defined on equivalence classes. For a well-defined function, there will be no ambiguity.

This section presents the most important components of the formalisation, omitting obvious definitions and routine proofs. The approach should work with tools such as HOL [Gordon and Melham 1993] and PVS [Crow et al. 1995]. Isabelle output is shown, using mathematical symbols that are largely self-explanatory. Note that the Isabelle statement $[\![P_1; \ldots; P_n]\!] \Rightarrow Q$ is equivalent to $P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow Q$ and denotes the inference rule $\frac{P_1 \ \ldots \ P_n}{Q.}$

Another notation in need of explanation is the image operator. A relation in Isabelle/HOL is a set of ordered pairs. The notation $R``A$ denotes the image of the set $A$ under the relation $R$, namely $\{y \mid \exists x \in A.\ (x,y) \in R\}$. In particular, $R``\{x\}$ denotes $\{y \mid (x,y) \in R\}$, which is the equivalence class $[x]_R$ when $R$ is an equivalence relation. (HOL users traditionally represent the relation $R$ as a function of type $\alpha \to \alpha \to \texttt{bool}$. They can express the equivalence class $[x]_R$ by currying: $R\,x$. Either approach is acceptable.)

The predicate `equiv A r` denotes that `r` is an equivalence relation on the set `A`. This concept is easily defined and its basic properties proved. Let us therefore focus on the theorems that are directly relevant to working with quotient types. This theorem, concerning equality between equivalence classes, is useful in congruence proofs.

**theorem** *eq_equiv_class_iff:*

```
"⟦equiv A r;  x ∈ A;  y ∈ A⟧
 ⟹ (r``{x} = r``{y}) = ((x,y) ∈ r)"
```

The quotient set `A//r` is defined to be the set of equivalence classes.

```
"A//r ≡ ⋃x ∈ A. {r``{x}}"
```

The combination of unions and singleton sets in this definition captures the concept of a set comprehension. I have formalized $\{f(x) \mid x \in A\}$ by $\bigcup_{x \in A} \{f(x)\}$. This technique generalizes nicely:

$$\{f(x_1, \ldots, x_n) \mid x_1 \in A_1, \ldots, x_n \in A_n\} = \bigcup_{x_1 \in A_1} \ \ldots \bigcup_{x_n \in A_n} \{f(x_1, \ldots, x_n)\}.$$

I use the same technique below when defining functions over equivalence classes. Nested unions of singleton sets work well. If an alternative formalization is chosen, then the lemmas proved below about unions must be reformulated accordingly.

Crucial to the method is the treatment of congruence properties. The formula `f respects r` expresses[2] that the function `f` respects the relation `r`. That is, `f` returns equal results for arguments that are related by `r`.

```
f respects r ≡ ∀y z. (y,z) ∈ r ⟶ f y = f z
```

This definition does not constrain the domain and range of `f`, but in the lemmas involving unions, this function must range over sets. In practice, these sets will either be singletons or equivalence classes. In untyped set theory, where everything is a set, they could be anything.

[2]Suggested by Rob Arthan. Previously, I used a less intuitive notation, `congruent r f`.

The next lemma expresses the central idea of the approach, although users do not invoke it directly. It eliminates unions over constant functions. For all values in its domain (namely `A`), the function `f` returns the set `c`.

**lemma** `"⟦a ∈ A; ∀y ∈ A. f y = c⟧ ⟹ (⋃y ∈ A. f y) = c"`

The following lemma is again crucial. It eliminates a union over the elements of an equivalence class provided the function respects the equivalence relation. Such unions will appear in the definitions of functions over quotient types. The removal of the union is precisely the simplification we need to obtain natural reasoning.

**lemma** *UN_equiv_class:*
```
"⟦equiv A r; f respects r; a ∈ A⟧
 ⟹ (⋃x ∈ r``{a}. f x) = f a"
```

For two-argument functions, we have the predicate *congruent2*, where *congruent2 r1 r2 f* means the function `f` respects the relations *r1* and *r2*.

```
∀x y u v. (x,y) ∈ r1 ⟶ (u,v) ∈ r2 ⟶ f x u = f y v
```

The intuitive syntax *f respects2 r* abbreviates the common situation when the two relations are identical, *congruent2 r r f*.

The congruence property for a two-argument function can be shown directly from the definition. Occasionally, it is easier to show *congruent2 r1 r2 f* by showing that `f` respects each relation separately. It also suffices to show that `f` is commutative and respects a relation in one argument. These straightforward lemmas, based on Harrison's HOL formalization, are omitted.

Here is a lemma to eliminate unions over the elements of equivalence classes, this time for two arguments.

**lemma** *UN_equiv_class2:*
```
"⟦equiv A r1; equiv A r2; congruent2 r1 r2 f; a1 ∈ A; a2 ∈ A⟧
 ⟹ (⋃x1 ∈ r1``{a1}. ⋃x2 ∈ r2``{a2}. f x1 x2) = f a1 a2"
```

It follows from the one argument case. The proof uses two lemmas concerning functions that satisfy *congruent2 r1 r2 f*. The first lemma asserts that the one-argument function `f a` obtained by currying, where `a ∈ A`, respects *r2*. The second lemma asserts that ⋃*x2 ∈ r2``{a2}. f x1 x2* respects *r1* when viewed as a function of *x1*.

One further lemma is helpful. Isabelle [Nipkow et al. 2002, §8.2.1] and HOL both allow a tuple of bound variables to appear wherever a bound variable is expected. They are translated into a primitive un-curry operator. For example, $\bigcup_{(x,y)\in A} Bxy$ abbreviates $\bigcup_{z\in A} (\lambda(x,y).\, Bxy)z$, where $\lambda(x,y).\, Bxy$ satisfies the equation

$$(\lambda(x,y).\, Bxy)(X,Y) = BXY.$$

The lemma rewrites the left-hand side (which users may employ in definitions) into nested unions, a form that allows the application of the union lemma.

**lemma** *UN_UN_split_split_eq:*
```
"(⋃(x1,x2) ∈ X. ⋃(y1,y2) ∈ Y. A x1 x2 y1 y2) =
 (⋃x∈X. ⋃y∈Y. (λ(x1,x2). (λ(y1,y2). A x1 x2 y1 y2) y) x)"
```

The lemma is unnecessary for people who are willing to use the projection functions `fst` and `snd` (`PROJ.1` and `PROJ.2` in PVS). It is specific to the common case where the representing set `A` consists of ordered pairs.

## 4. EXAMPLE: DEFINING THE INTEGERS FORMALLY

This section applies the lemma library to a formal development of the integers. The technique is independent of particular theorem provers. Isabelle's approach to type definition (which resembles HOL's) clutters some of the formulae with abstraction and representation functions. These will be absent in PVS or in any untyped formalism.

We also need a function `contents` that returns the sole element of a singleton set. It must satisfy the equation `contents {x} = x`. In higher-order logic, defining such a function requires a definite description (the $\iota$-operator). It does not require an indefinite description (Hilbert's $\epsilon$-operator) because we are only interested in singleton sets.[3]

Standard set theories—untyped and without atoms—satisfy $\bigcup(\{x\}) = x$. We could therefore use $\bigcup$ for `contents`, but we can do better still. Both `contents` and singletons (the inner $\{\cdots\}$) can be eliminated from function definitions by virtue of the equation

$$\bigcup\left(\bigcup_{x\in A}\{f(x)\}\right) = \bigcup_{x\in A} f(x).$$

Even in higher-order logic, this simplification can be used to define some functions whose result involves a set construction. Indeed, `contents` and singletons can be removed from most of the definitions below. This section uses the general form in order to demonstrate the general technique; the next section will use simplified definitions.

### 4.1 Defining the Integers in Isabelle/HOL

We begin by defining the equivalence relation.

```
"intrel ≡ {((x,y),(u,v)) | x y u v. x+v = u+y}"
```

Next, we introduce the type int.

```
typedef (Integ)  int = "UNIV//intrel"
  by (auto simp add: quotient_def)
```

An Isabelle type definition equates a new type to a set. The command **by** `(auto ...)` proves the set to be non-empty by appealing to the definition of quotients. The set is given the name `Integ` and is defined to be the set of pairs of natural numbers quotiented by `intrel`. The polymorphic constant `UNIV` denotes the universal set, here of type `nat*nat`. The representation function maps elements of type `int` to elements of type `nat*nat` belonging to the set `Integ`; the abstraction function maps in the opposite direction.

Now we can define the integer constants 0 and 1.

[3]The Isabelle tutorial discusses description operators [Nipkow et al. 2002, §5.10].

```
"0 ≡ Abs_Integ(intrel `` {(0,0)})"
"1 ≡ Abs_Integ(intrel `` {(1,0)})"
```

Here `intrel `` {(0,0)}` denotes the equivalence class $[(0,0)]$, which `Abs_Integ` coerces to the new type `int`.

Now, we can do some preliminary proofs. A trivial equivalence is proved and given to the simplifier (through the `[simp]` annotation). It unfolds membership assertions concerning the equivalence relation without unfolding the definition in other contexts.

```
lemma intrel_iff[simp]: "(((x,y),(u,v)) ∈ intrel) = (x+v = u+y)"
by (simp add: intrel_def)
```

Proving that `intrel` is an equivalence relation requires a one-line simplifier call. (`UNIV` again denotes the universal set of type `nat*nat`.)

```
lemma equiv_intrel: "equiv UNIV intrel"
by (simp add: intrel_def equiv_def refl_def sym_def trans_def)
```

A few other routine declarations (omitted) complete the setup of type `int`.

Reasoning about the integers requires a theorem stating that every integer is represented by a pair of natural numbers. This theorem is trivially expressed in first-order logic by $\forall z\ \exists xy\ z = [(x,y)]$. In proofs, this theorem replaces given integer variables by equivalence classes involving arbitrary natural numbers. Isabelle's natural deduction framework can express this reasoning step directly as an inference rule.

```
lemma eq_Abs_Integ [cases type: int]:
    "(⋀x y. z = Abs_Integ(intrel``{(x,y)}) ⟹ P) ⟹ P"
```

The annotation `cases type: int` informs Isabelle that case analysis on an integer variable—through the `cases` command—refers to this rule implicitly. This Isabelle-specific feature should not be difficult to imitate using other tools. In HOL, it is easy to write a tactic that takes an integer variable, creates a suitable instance of the theorem, and eliminates the existential quantifiers.

### 4.2 A One-Argument Function on Equivalence Classes

Unary minus illustrates the definition of functions on quotient types. The idea is simple. We cannot pick an arbitrary element of an equivalence class, but if the function respects the equivalence relation, then the choice of element does not matter. Therefore, we form a set consisting of all values generated by all elements of the equivalence class. This set will simplify to a singleton, whose value will be returned via the equation `contents {x} = x`.

```
"-z ≡ contents
      (⋃(x,y)∈Rep_Integ z. { Abs_Integ(intrel``{(y,x)}) })"
```

Here `intrel `` {(y,x)}` denotes the equivalence class $[(y,x)]$. The argument of `contents` is the collection of all integers $[(y,x)]$ such that $(x,y)$ belongs to the equivalence class for $z$. This collection will turn out to be a singleton.

4.2.1 *Proving the Characteristic Equation.* Let us apply these ideas to integer negation. We can prove its *characteristic equation* $-[(x,y)] = [(y,x)]$, describing its behaviour on equivalence classes.

```
lemma minus:
  "- Abs_Integ(intrel``{(x,y)}) = Abs_Integ(intrel `` {(y,x)})"
proof -
  have "(λ(x,y). {Abs_Integ (intrel``{(y,x)})}) respects intrel"
    by (simp add: congruent_def)
  thus ?thesis
    by (simp add: minus_int_def UN_equiv_class [OF equiv_intrel])
qed
```

The first part of the proof concerns congruence: that of the body of the union in the definition of negation. The simplifier, given the definition of congruence (`congruent_def`), immediately establishes that claim. In general, congruence properties can be difficult to prove.

The second part of the proof establishes the desired equation using the definition of negation (`minus_int_def`) and our theorem about unions over equivalence classes. This part of the proof is always easy. Above, `UN_equiv_class [OF equiv_intrel]` denotes the instance of `UN_equiv_class` for the relation `intrel`; since we have just proved the necessary congruence property, it is in effect the following rewrite rule:

```
"(⋃x ∈ intrel``{a}. f x) = f a"
```

Using this rule, the left-hand side of the desired equation simplifies immediately to

```
"contents ({ Abs_Integ(intrel``{(y,x)}) })"
```

Then, because `contents` is applied to a singleton, this simplifies in one step to the desired right-hand side. Even the most basic rewriting engine can perform the reasoning outlined above.

4.2.2 *An Alternative: the Axiom of Choice.* The obvious way to formalize equivalence classes, used by virtually all other researchers, employs Hilbert's $\epsilon$-operator to choose a representative of each equivalence class. In that approach, negation might be defined as follows:

```
"-z ≡ (let (x,y) = choose (Rep_Integ z) in Abs_Integ(intrel``{(y,x)}))"
```

Here, `choose` is a function that returns an arbitrary element of a given set. This reliance on the axiom of choice does not lessen the proof obligation: we must still show that the function respects the equivalence relation. Reasoning about the axiom of choice will be difficult unless we can find lemmas resembling `UN_equiv_class` to eliminate the operator `choose`. Even if we can, the formalization involving unions is preferable because it avoids a needless dependence on the axiom of choice.

4.2.3 *Reasoning about the Newly Defined Function.* Given the characteristic equation, proving properties of unary negation is trivial. The approach is always the same. First, let the `cases` command replace integer variables by equivalence classes. Then, call the simplifier to replace integer constants (such as 0) by their definitions and to apply the characteristic equation and other simplification rules. At this point, the formal proof steps duplicate those of textbook proofs.

Consider the proof that negation is self-cancelling.

```
lemma zminus_zminus: "- (- z) = (z::int)"
apply (cases z)
apply (simp add: minus)
done
```

The *cases* command leaves the following proof state:

```
1. ⋀x y. z = Abs_Integ (intrel `` {(x, y)}) ⟹ - (- z) = z
```

The simplifier uses the characteristic equation (*minus*) twice, each time exchanging the two variables. The machine proof follows the informal one presented in §2 above.

```
- (- (Abs_Integ (intrel `` {(x, y)})))
  = - (Abs_Integ (intrel `` {(y, x)}))
  = Abs_Integ (intrel `` {(x, y)})
```

### 4.3 Two-Argument Functions on Equivalence Classes

Addition and multiplication illustrate the treatment of two-argument functions. There are simply two unions instead of one.

```
"z + w ≡
    contents (⋃(x,y)∈Rep_Integ z. ⋃(u,v)∈Rep_Integ w.
              { Abs_Integ(intrel``{(x+u, y+v)}) })"

"z * w ≡
    contents (⋃(x,y)∈Rep_Integ z. ⋃(u,v)∈Rep_Integ w.
              { Abs_Integ(intrel``{(x*u + y*v, x*v + y*u)}) })"
```

The characteristic equation for addition describes its effect on equivalence classes: $[(x,y)]+[(u,v)] = [(x+u, y+v)]$. The proof again begins by establishing congruence. Then the main theorem is established using the definition of addition and our union theorems for two-argument functions. As with unary negation above, the proof is a short and simple equational argument.

```
lemma add:
   "Abs_Integ (intrel``{(x,y)}) + Abs_Integ (intrel``{(u,v)}) =
    Abs_Integ (intrel``{(x+u, y+v)})"
proof -
  have "(λz w. (λ(x,y). (λ(u,v).
        {Abs_Integ (intrel `` {(x+u, y+v)})}) w) z)
       respects2 intrel"
    by (simp add: congruent2_def)
  thus ?thesis
    by (simp add: add_int_def UN_UN_split_split_eq
                  UN_equiv_class2 [OF equiv_intrel])
qed
```

The congruence property, `... respects2 intrel`, looks formidable. However, manually applying the theorem *UN_equiv_class2* displays the required claim, which we can then paste into the proof script. The arithmetic decision procedure for the natural numbers makes the proof trivial.

Consider the proof that unary minus distributes over addition.

```
lemma zminus_zadd_distrib: "- (z + w) = (- z) + (- w::int)"
apply (cases z, cases w)
apply (simp add: minus add)
done
```

Here is the proof state after both applications of the `cases` method.

```
1. ⋀x y xa ya.
      ⟦z = Abs_Integ (intrel `` {(x, y)});
       w = Abs_Integ (intrel `` {(xa, ya)})⟧
      ⟹ - (z + w) = - z + - w
```

This subgoal is proved by rewriting using the characteristic equations for negation and addition. The formal reasoning is essentially the same as the textbook proof.

The treatment of multiplication is similar. The proof of congruence (omitted) requires some work because it lies outside the scope of linear arithmetic. Given that lemma, simple equational reasoning (as always) establishes the characteristic equation:

```
lemma mult:
  "Abs_Integ((intrel``{(x,y)})) * Abs_Integ((intrel``{(u,v)})) =
   Abs_Integ(intrel `` {(x*u + y*v, x*v + y*u)})"
by (simp add: mult_int_def UN_UN_split_split_eq mult_congruent2
              UN_equiv_class2 [OF equiv_intrel equiv_intrel])
```

We can now prove the standard theorems relating negation, addition and multiplication. Each proof consists of `cases` followed by simplification with characteristic equations and the corresponding properties of the natural numbers. Other proofs, not shown, are equally trivial.

```
lemma zmult_zminus: "(- z) * w = - (z * (w::int))"
by (cases z, cases w, simp add: minus mult add_ac)

lemma zmult_assoc: "((z1::int) * z2) * z3 = z1 * (z2 * z3)"
by (cases z1, cases z2, cases z3,
    simp add: mult add_mult_distrib2 mult_ac)

lemma zmult_commute: "(z::int) * w = w * z"
by (cases z, cases w, simp add: mult add_ac mult_ac)

lemma zadd_zmult_distrib: "((z1::int)+z2) * w = (z1*w) + (z2*w)"
by (cases z1, cases z2, cases w,
    simp add: add mult add_mult_distrib2 mult_ac)
```

### 4.4 Further Operations on the Integers

The treatment of the ordering ($\leq$) illustrates an advantage of this approach. A relation in higher-order logic is a Boolean-valued function, so we could have used nested unions as we did for addition and multiplication. However, because we are using native logic rather than a package, we are not forced to formalise this relation as a function.

```
  "z ≤ (w::int)
   ≡ ∃x y u v. x+v ≤ u+y &
               (x,y) ∈ Rep_Integ z & (u,v) ∈ Rep_Integ w"
```

We can prove the characteristic equation directly, without proving congruence. The proof is trivial. Informally, the equation is $[(x,y)] \leq [(u,v)] \iff x+v \leq u+y$.

```
lemma le:
  "(Abs_Integ(intrel``{(x,y)}) ≤ Abs_Integ(intrel``{(u,v)}))
   = (x+v ≤ u+y)"
by (force simp add: le_int_def)
```

The proofs about the ordering are largely straightforward, and are therefore omitted. The only difficult one is the monotonicity of multiplication, and it would be no easier using other treatments of quotient types.

A final example is the coercion from integers to natural numbers. It illustrates a function that leaves the integers.

```
"nat z ≡ contents (⋃(x,y) ∈ Rep_Integ z. { x-y })"
```

This function respects the equivalence relation, and its characteristic equation is $\mathrm{nat}[(x,y)] = x-y$. Note that $x-y$ is natural number subtraction and that $x-y = 0$ if $x \leq y$.

```
lemma nat: "nat (Abs_Integ (intrel``{(x,y)})) = x-y"
proof -
  have "(λ(x,y). {x-y}) respects intrel"
    by (simp add: congruent_def, arith)
  thus ?thesis
    by (simp add: nat_def UN_equiv_class [OF equiv_intrel])
qed
```

Using this characteristic equation, theorems relating the function `nat` to the other integer operations are trivial to prove.

## 5. QUOTIENTING A RECURSIVE DATA TYPE

Another application of equivalence relations is to impose equations on recursive datatypes. The necessary declarations are voluminous, but they are not complicated and can be produced with the assistance of cut-and-paste. This section presents an example inspired by cryptographic protocols. Many symmetric-key cryptosystems provide separate decryption and encryption operations, which are inverses of each other. Writing decryption of message $X$ using key $K$ as $D_K(X)$ and the corresponding encryption as $E_K(X)$, we have the equations $D_K(E_K(X)) = X$ and $E_K(D_K(X)) = X$. Decryption can be applied to any message, not just to the result of an encryption.

To define a datatype with equational constraints, first define an ordinary datatype (which will be a free algebra). Then, define an equivalence relation expressing the desired equations; the precise form is illustrated below. Finally, quotient the datatype. The free datatype constructors are easily lifted to the new recursive datatype, using the techniques of function definition described above. To define other functions on the new datatype, first define a concrete version on the free datatype and then lift it.

### 5.1 The Concrete Datatype and the Equivalence Relation

This simple datatype has four constructors: a message can be a nonce (a number), the concatenation of two messages, an encryption or a decryption.

```
datatype
     freemsg = NONCE  nat
             | MPAIR  freemsg freemsg
             | CRYPT  nat freemsg
             | DECRYPT  nat freemsg
```

The equivalence relation, *msgrel*, is defined inductively. The first two rules (*CD* and *DC*) express the desired equations between encryption and decryption. The next four rules (*NONCE* to *DECRYPT*) have many purposes. They make the equations hold for sub-messages; they allow the abstract constructors to respect *msgrel*; they ensure that *msgrel* is reflexive. The last two rules (*SYM* and *TRANS*) ensure that *msgrel* is symmetric and transitive.

```
inductive "msgrel"
  intros
    CD:    "CRYPT K (DECRYPT K X) ∼ X"
    DC:    "DECRYPT K (CRYPT K X) ∼ X"
    NONCE: "NONCE N ∼ NONCE N"
    MPAIR: "⟦X ∼ X'; Y ∼ Y'⟧ ⟹ MPAIR X Y ∼ MPAIR X' Y'"
    CRYPT: "X ∼ X' ⟹ CRYPT K X ∼ CRYPT K X'"
    DECRYPT: "X ∼ X' ⟹ DECRYPT K X ∼ DECRYPT K X'"
    SYM:   "X ∼ Y ⟹ Y ∼ X"
    TRANS: "⟦X ∼ Y; Y ∼ Z⟧ ⟹ X ∼ Z"
```

The relation $\sim$ is easily proved to be reflexive, symmetric and transitive. The proof of *X*$\sim$*X* is by structural induction on the message *X*.

### 5.2 Two Functions on the Free Algebra

Two examples will illustrate how functions are lifted to the quotiented abstract datatype. Obviously, we can only consider functions that respect the equivalence relation. Both of these functions ignore encryption and decryption altogether, so they are acceptable.

The function *freenonces* returns the set of all nonces present in a message. It is defined by structural recursion.

```
  "freenonces (NONCE N) = {N}"
  "freenonces (MPAIR X Y) = freenonces X ∪ freenonces Y"
  "freenonces (CRYPT K X) = freenonces X"
  "freenonces (DECRYPT K X) = freenonces X"
```

This function respects the equivalence relation. The one-line proof appeals to induction on the definition of $\sim$ followed by simplification.

```
theorem msgrel_imp_eq_freenonces:
    "U ∼ V ⟹ freenonces U = freenonces V"
by (erule msgrel.induct, auto)
```

The function *freeleft* returns the left part of the topmost *MPAIR* constructor. (The lifted version, *left*, will be a destructor function for the abstract *MPair* constructor.) The cases for *CRYPT* and *DECRYPT* make it respect the equivalence relation. The case for *NONCE* makes the function total, and will yield the further equation *left (Nonce N) = Nonce N*.

```
  "freeleft (NONCE N) = NONCE N"
  "freeleft (MPAIR X Y) = X"
  "freeleft (CRYPT K X) = freeleft X"
  "freeleft (DECRYPT K X) = freeleft X"
```

The proof that *freeleft* respects the equivalence relation resembles the previous one, but includes an appeal to *msgrel.intros*: a list of theorems for proving membership in $\sim$.

```
theorem "U ∼ V ⟹ freeleft U ∼ freeleft V"
by (erule msgrel.induct, auto intro: msgrel.intros)
```

### 5.3 The Abstract Message Type and its Constructors

The abstract type of messages is declared by quotienting the universal set (here of type *freemsg*) with the relation $\sim$.

```
typedef (Msg)  msg = "UNIV//msgrel"
  by (auto simp add: quotient_def)
```

The abstract versions of the message constructors are called *Nonce*, *MPair*, *Crypt* and *Decrypt*. They are defined as functions using unions, as we have already seen for the integer operations. The following definitions do not use the function *contents*; this simplification is possible because the results are all derived from a set (namely an equivalence class).

```
"Nonce N == Abs_Msg(msgrel``{NONCE N})"

"MPair X Y ==
  Abs_Msg (⋃U∈Rep_Msg X. ⋃V∈Rep_Msg Y. msgrel``{MPAIR U V})"

"Crypt K X == Abs_Msg (⋃U∈Rep_Msg X. msgrel``{CRYPT K U})"

"Decrypt K X == Abs_Msg (⋃U∈Rep_Msg X. msgrel``{DECRYPT K U})"
```

Proving the characteristic equations for these constructors is straightforward. Each equation relates an abstract constructor to the corresponding concrete constructor. Each congruence proof is immediate by the definition of the equivalence relation. Recall that the proof of the characteristic equation from the congruence property is trivial.

```
"MPair (Abs_Msg(msgrel``{U})) (Abs_Msg(msgrel``{V})) =
 Abs_Msg (msgrel``{MPAIR U V})"

"Crypt K (Abs_Msg(msgrel``{U})) = Abs_Msg(msgrel``{CRYPT K U})"
```

There is no characteristic equation for *Nonce* because its argument is not an equivalence class. The characteristic equation for *Decrypt* appears below, with its proof. As always, congruence is established first.

```
lemma Decrypt:
     "Decrypt K (Abs_Msg(msgrel``{U})) =
      Abs_Msg (msgrel``{DECRYPT K U})"
proof -
  have "(λU. msgrel `` {DECRYPT K U}) respects msgrel"
```

```
    by (simp add: congruent_def msgrel.DECRYPT)
  thus ?thesis
    by (simp add: Decrypt_def UN_equiv_class [OF equiv_msgrel])
qed
```

As with the integers, the *cases* lemma lets us replace an abstract message by its representation as an equivalence class.

```
lemma eq_Abs_Msg [cases type: msg]:
     "(!!U. z = Abs_Msg(msgrel``{U}) ==> P) ==> P"
```

We now achieve a key objective: *Crypt* and *Decrypt* are indeed inverses. Both proofs are one-liners using *cases*, the characteristic equations and the corresponding rule from the inductive definition of $\sim$.

```
theorem CD_eq: "Crypt K (Decrypt K X) = X"
by (cases X, simp add: Crypt Decrypt CD)

theorem DC_eq: "Decrypt K (Crypt K X) = X"
by (cases X, simp add: Crypt Decrypt DC)
```

Both proofs implicitly refer to the theorem *eq_equiv_class_iff*, presented in §3 above, which relates equality of equivalence classes to membership in the equivalence relation.

### 5.4 Defining Functions on the Abstract Message Type

To define a function on the abstract message type, first define an analogous version on the concrete type and prove that it respects equivalence relation. Then, define the abstract version as usual using unions.

Recall that the function *freenonces* returns the set of nonces contained in a concrete message. We are now ready to declare the corresponding abstract function.

```
  "nonces X == ⋃U∈Rep_Msg X. freenonces U"
```

Congruence is immediate since *freenonces* respects the equivalence relation, as we saw in §5.2 above.

```
lemma nonces_congruent: "freenonces respects msgrel"
by (simp add: congruent_def msgrel_imp_eq_freenonces)
```

The recursion equations for *nonces*, the abstract function, are trivial to prove. Here are the first three.

```
"nonces (Nonce N) = {N}"
"nonces (MPair X Y) = nonces X ∪ nonces Y"
"nonces (Crypt K X) = nonces X"
```

Here is the fourth equation, including its proof. Observe its similarity to proofs about the integer operations.

```
lemma nonces_Decrypt: "nonces (Decrypt K X) = nonces X"
apply (cases X)
apply (simp add: nonces_def Decrypt
                 UN_equiv_class [OF equiv_msgrel nonces_congruent])
done
```

Recall that the function *freeleft* returns returns the left part of the topmost pair in a concrete message. The abstract version is defined using the techniques just demonstrated.

```
"left X == Abs_Msg (⋃U∈Rep_Msg X. msgrel``{freeleft U})"
```

Here are the recursion equations for this abstract destructor function.

```
"left (Nonce N) = Nonce N"
"left (MPair X Y) = X"
"left (Crypt K X) = left X"
"left (Decrypt K X) = left X"
```

### 5.5 Freeness of the Abstract Constructors

The abstract datatype satisfies equations between some of its constructors, but the other constructors are still injective. In my experience, the easiest way to prove such properties is to define explicit destructor functions, not by inductive proofs over equivalence relations. We have seen the function *left* above, and the function *right* can be defined similarly. These two functions make it easy to prove that abstract message pairing is injective.

```
"(MPair X Y = MPair X' Y') = (X=X' & Y=Y')"
```

The function *nonces* makes it easy to prove that the abstract nonce constructor is injective.

```
"(Nonce m = Nonce n) = (m = n)"
```

Surprising as it may seem, the constructors for encryption and decryption are also injective in the second argument; the proof is trivial, by the equations relating them.

```
"(Crypt K X = Crypt K X') = (X=X')"
"(Decrypt K X = Decrypt K X') = (X=X')"
```

They are not injective in the first argument because any abstract message can be rewritten to have the form of an encryption or decryption with any desired key. For the same reason, we do not have the discrimination property *Nonce N* ≠ *Crypt K X*. However, many discrimination properties do hold, even for *Crypt* and *Decrypt*. They can be proved by defining a discriminator function on the concrete datatype, using the methods demonstrated above.

```
"freediscrim (NONCE N) = 0"
"freediscrim (MPAIR X Y) = 1"
"freediscrim (CRYPT K X) = freediscrim X + 2"
"freediscrim (DECRYPT K X) = freediscrim X - 2"
```

Because encryptions and decryptions cancel each other, this function respects the equivalence relation. Lifting this function to the quotiented type yields a function *discrim* satisfying the four analogous equations on the abstract message constructors. Thus we can prove that no nonce equals a message pair.

```
"Nonce N ≠ MPair X Y"
```

We can also prove many discrimination results involving encryption, such as this one.

```
"Crypt K (Nonce M) ≠ Nonce N"
```

We have seen just one example of a quotiented datatype. Imposing equations on the constructors seems to be straightforward, using the strategy shown above. Establishing other properties, such as inequations, requires defining suitable functions on the free datatype.

At a referee's request, I have formalized an example involving nested recursion: a quotiented datatype of expressions, including a constructor to apply a function to a list of expressions. This development (available upon request) uses a function to form a equivalence relation on lists from an equivalence relation on list elements. It also requires a `cases` lemma analogous to `eq_Abs_Msg`, but for lists of expressions. It is clear that handling the full range of quotient datatypes requires considerable ingenuity.

## 6. RELATED WORK AND CONCLUSIONS

The aim of this paper is to demystify the process of defining functions over equivalence classes. Other authors have worked in this field.

Harrison [1994, §5] has written an automated package for HOL that declares the abstract quotient type and operations, returning theorems about those operations. Its arguments include the equivalence relation, the desired characteristic equations for each operation, and proofs of each theorem expressed at the level of representatives. However, such a package is not essential. The necessary definitions are straightforward, if they are written as described above, and the reasoning about equivalence classes poses no difficulties. Not using a package has its advantages: we do not have to collect all the theorems we shall ever want into one giant list; we are not restricted to top-level properties but can reason about equivalence classes within a larger proof; we do not get stuck because the package developer failed to anticipate our special requirements.

For defining abstract datatypes, the situation is different. Each constructor and destructor function requires a separate declaration and congruence proof. The declarations are uniform and the proofs are trivial, but there are too many of them. Who would like to see the full treatment of a 20 constructor abstract datatype? Nested and mutual recursion are also difficult. If such declarations on needed frequently, the proof tool should provide automated support.

Homeier [2001] has developed an elaborate package for HOL that offers special support for quotient constructions on recursive data types. Homeier has used it to define a type of $\lambda$-terms quotiented under $\alpha$-equivalence. Wenzel's treatment of quotient types in Isabelle uses axiomatic type classes, which streamlines the notation but tends to require additional type declarations.[4] Slotosch [1997] has developed an Isabelle/HOL theory of higher-order quotients based on partial equivalence relations.

PVS supports quotients in its standard prelude, but not with dedicated code. Jackson has done a few examples, including a construction of the integers (pri-

[4]See the theory at `http://isabelle.in.tum.de/library/HOL/Library/Quotient.html`.

vate communication), while Tews [2004] has formalized one of the theorems in a theoretical paper on coalgebras.

All of these other treatments of quotients use the axiom of choice, typically via Hilbert's $\epsilon$-operator, to pick arbitrary elements of equivalence classes. However, using the axiom of choice does not lessen the proof obligations. Pragmatists may argue that in verification nobody cares whether choice is used or not. However, pragmatists should be concerned that reasoning about $\epsilon$-terms is tricky, while the unions over equivalence classes are simplified away automatically.

We have seen, through a series of simple examples, how to define functions on equivalence classes and how to reason about them. No special tools are required, only a small lemma library. Each function definition must be expressed in a particular form. Provided it respects the equivalence relation, its characteristic equation is easily proved. Properties of the function can then be reduced to properties of the representing type, with proofs that resemble textbook presentations.

## ACKNOWLEDGMENTS

Rob Arthan, Francis Flannery, Tom Harke, Farhad Mehta, Lockwood Morris, Markus Wenzel and the referees commented on this paper. Peter Homeier provided extensive information about his tools. The U.K.'s Engineering and Physical Sciences Research Council (EPSRC) supported the development of Isabelle.

## REFERENCES

Crow, J., Owre, S., Rushby, J., Shankar, N., and Srivas, M. 1995. A tutorial introduction to PVS. Tech. rep., Computer Science Laboratory, SRI International.

de Bruijn, N. G. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indagationes Mathematicae 34*, 381–392.

Geuvers, H., Pollack, R., Wiedijk, F., and Zwanenburg, J. 2002. A constructive algebraic hierarchy in Coq. *Journal of Symbolic Computation 34,* 4, 271–286.

Gordon, M. J. C. and Melham, T. F. 1993. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

Harrison, J. 1994. Constructing the real numbers in HOL. *Formal Methods in System Design 5*, 35–59.

Homeier, P. V. 2001. Quotient types. In *TPHOLs 2001: Supplemental Proceedings*, R. J. Boulton and P. B. Jackson, Eds. Number EDI-INF-RR-0046 in Informatics Report Series. Division of Informatics, University of Edinburgh, 191–206. Online at `http://www.informatics.ed.ac.uk/publications/report/0046.html`.

Kaufmann, M. and Moore, J. S. 2002. *ACL2 Version 2.7*. University of Texas at Austin. On the Internet at `http://www.cs.utexas.edu/users/moore/acl2/v2-7/`.

Kolman, B., Busby, R. C., and Ross, S. C. 2000. *Discrete Mathematical Structures*, 4th ed. Prentice-Hall.

Nipkow, T., Paulson, L. C., and Wenzel, M. 2002. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer. LNCS Tutorial 2283.

Slotosch, O. 1997. Higher order quotients and their implementation in Isabelle HOL. In *Theorem Proving in Higher Order Logics: TPHOLs '97*, E. L. Gunter and A. Felty, Eds. LNCS 1275. Springer, 291–306.

Tews, H. 2004. Predicate and relation lifting for parametric algebraic specifications. In *CMCS 2004, Seventh Workshop on Coalgebraic Methods in Computer Science*, J. Adámek, Ed. Electronic Notes in Theoretical Computer Science. 361–378.

Received April 2004; revised September 2004; accepted September 2004